Quaderni del Dipartimento di Matematica
Università degli Studi di Parma

Elio Panegai and Gianfranco Rossi[1] (Eds.)

Proceedings of CILC'04
Italian Conference on Computational Logic

November 16, 2004 n. 390

[1]Dipartimento di Matematica, Università di Parma, I-43100 Parma, Italy, gianfr@prmat.math.unipr.it

Gianfranco Rossi   Elio Panegai   (Eds.)

CILC'04

ITALIAN CONFERENCE ON COMPUTATIONAL LOGIC

Nineteenth Annual Meeting of GULP - (Associazione Italiana Gruppo Ricercatori e Utenti di Logic Programming)

June 16-17, 2004
Parma, Italy

PROCEEDINGS

Preface

This report contains the Proceedings of the *CILC'04 Italian Conference on Computational Logic* held in Parma (Italy), June 16-17, 2004. This is the 19th annual meeting of the Italian Association *"Gruppo Ricercatori e Utenti di Logic Programming" (GULP)*. Previous GULP Conferences were held in Italy (from 1986 to 1993), and, since 1994, alternatively in Italy, Spain, Portugal (and Cuba, in 2000), in cooperation with the Spanish PRODE Association and the Portuguese Association for Artificial Intelligence APPIA. The 2004 edition changed its name to CILC in order to emphasize the Computational Logic issue, and it moved again to a more "local" organization structure (anyway, still open to contributions from everywhere and with a wide scope of interests).

The technical program of the Conference includes 25 communications, organized according to the following topics: Abductive Logic Programming, Automated Reasoning, Constraints, Data Management, Learning and Knowledge Discovery, Logic-Based Agents, Program Refinement and Transformation, Semantics. All papers have been evaluated by two reviewers. In addition, the program includes an invited talk by Luís Moniz Pereira (U. Lisbon), three tutorials by Stefania Costantini (U. L'Aquila), Agostino Dovier (U. Udine) and Paolo Torroni (U. Bologna), and four "demo" presentations.

We wish to thanks all authors, the invited speaker, the tutorialists, the members of the Program Committee, all participants, and everyone who contributed to the success of the Conference.

June 2004

Elio Panegai
Gianfranco Rossi
Editors

Officials

Program Committee

Conference Chairman
Gianfranco Rossi (Univ. di Parma)

Matteo Baldoni (Univ. di Torino)
Francesco Buccafurri (Univ. "Mediterranea" di Reggio Calabria)
Michele Bugliesi (Univ. "Ca Foscari" di Venezia)
Marco Cadoli (Univ. "La Sapienza" di Roma)
Stefania Costantini (Univ. di L'Aquila)
Giorgio Delzanno (Univ. di Genova)
Agostino Dovier (Univ. di Udine)
Andrea Formisano (Univ. di L'Aquila)
Marco Gavanelli (Univ. di Ferrara)
Fosca Giannotti (CNR, Pisa)
Roberta Gori (Univ. di Pisa)
Vincenzo Loia (Univ. di Salerno)
Donato Malerba (Univ. di Bari)
Maria Chiara Meo (Univ. "G. d'Annunzio" di Chieti e Pescara)
Andrea Omicini (Univ. di Bologna)
Maurizio Proietti (IASI-CNR, Roma)
Francesca Rossi (Univ. di Padova)
Fausto Spoto (Univ. di Verona)
Enea Zaffanella (Univ. di Parma)

Local Committee

Roberto Bagnara (Univ. di Parma)
Dario Bianchi (Univ. di Parma)
Elio Panegai (Univ. di Parma)
Cristina Reggiani (Univ. di Parma)
Gianfranco Rossi (Univ. di Parma)
Enea Zaffanella (Univ. di Parma)

Organized by

GULP, Gruppo Ricercatori e Utenti di Logic Programming.

With the contribution of:

Universitá degli Studi di Parma
Dipartimento di Matematica
CoLogNET

Contents

Invited Lecture

Revised Stable Models - a new semantics for logic programs 3
L. M. Pereira, A. M. Pinto

Tutorials

Answer Set Programming 7
S. Costantini

Il problema del Protein Folding e i relativi approcci basati su programmazione con vincoli 8
A. Dovier

Introduzione ai sistemi multi-agente basati su logica computazionale 9
P. Torroni

Contributed Talks

Abductive Logic Programming

Abduction with Hypotheses Confirmation 13
M. Alberti, M. Gavanelli, E. Lamma, P. Mello, P. Torroni

Abductive Logic Programming with CIFF: Implementation and Applications 28
U. Endriss, P. Mancarella, F. Sadri, G. Terreni, F. Toni

Automated Reasoning

A multi context-based Approximate Reasoning 43
L. Blandi, M. I. Sessa

A declarative approach to uncertainty orders 56
A. Capotorti, A. Formisano

SAT-based Analysis of Cellular Automata 73
M. D'Antonio, G. Delzanno

Uniform relational frameworks for modal inferences 88
A. Formisano, E. G. Omodeo, E. S. Orlowska, A. Policriti

Constraints

Constraint Based Protein Structure Prediction Exploiting Secondary Structure 103
R. Backofen, A. Dal Palú, A. Dovier, S. Will

Using a theorem prover for reasoning on constraint problems 118
M. Cadoli, T. Mancini

Constrained CP-nets 133
S. Prestwich, F. Rossi, K. B. Venable, T. Walsh

Data Management

Efficient Evaluation of Disjunctive Datalog Queries with Aggregate Functions DB 148
M. Citrigno, W. Faber, G. Greco, N. Leone

Checking the Completeness of Ontologies: A Case Study from the Semantic Web 163
V. Cordí, V. Mascardi

Combining logic programming and domain ontologies for text classification 178
C. Cumbo, S. Iiritano, P. Rullo

Ontological encapsulation of many-valued logic 189
Z. Majkic

Learning and Knowledge Discovery

Frequent Pattern Queries for Flexible Knowledge Discovery 202
F. Bonchi, F. Giannotti, D. Pedreschi

Costruzione automatica di courseware in DyLOG (short paper) 217
L. Torasso

Improving efficiency of recursive theory learning 220
A. Varlaro, M. Berardi, D. Malerba

Logic-Based Agents

From logic programs updates to action description updates 235
J. Alferes, F. Banti, A. Brogi

Reasoning About Logic-based Agent Interaction Protocols 250
M. Baldoni, C. Baroglio, A. Martelli, V. Patti

Contracts in Multiagent Systems: the Legal Institution Perspective 265
G. Boella, L. van der Torre

Integrating tuProlog into DCaseLP to Engineer Heterogeneous Agent Systems 280
I. Gungui, V. Mascardi

Communication Architecture in the DALI Logic Programming Agent-Oriented Language 295
S. Costantini, A. Tocchio, A. Verticchio

Programs Refinement and Trasformation

Preserving (Security) Properties under Action Refinement 310
A. Bossi, C. Piazza, S. Rossi

Totally Correct Logic Program Transformations Using Well-Founded Annotations (short paper) 325
A. Pettorossi, M. Proietti

Semantics

Implementing Joint Fixpoints Semantics on Top of DLV 330
F. Buccafurri, G. Caminiti

A compositional semantics for CHR 345
M. Gabbrielli, M. C. Meo

Demos

A demonstration of SOCS-SI 362
M. Alberti, F. Chesani, M. Gavanelli, E. Lamma, P. Mello, P. Torroni

Constraint-based tools for protein folding 365
L. Bortolussi, A. Dal Palú, A. Dovier

The DALI Logic Programming Agent-Oriented Language 368
S. Costantini, A. Tocchio

The JSetL library: supporting declarative programming in Java 372
E. Panegai, E. Poleo, G. Rossi

Index of Authors

Invited Lecture

Revised Stable Models - a new semantics for logic programs

Luís Moniz Pereira and **Alexandre Miguel Pinto**

Centro de Inteligência Artificial, Universidade Nova de Lisboa
2829-516 Caparica, Portugal
{lmp|amp}@di.fct.unl.pt

Abstract

This paper introduces an original 2-valued semantics for Normal Logic Programs (NLP), important on its own. Nevertheless, its name draws attention to that it is inspired by and generalizes Stable Model semantics (SM). The definitional distinction consists in the revision of one feature of SM, namely its treatment of odd loops over default negation. This single revised aspect, addressed by means of a *Reductio ad Absurdum* approach, affords us a fruitful cornucopia of consequences, namely regarding existence, relevance and top-down querying, cumulativity, and implementation.

The paper motivates and then defines the Revised Stable Models semantics (rSM), justifying the definition and providing examples. It also presents two rSM semantics preserving program transformations into NLP without odd loops. Properties of rSM are given and contrasted with those of SM. Implementation is examined, and extensions of rSM are given with regard to explicit negation, 'not's in heads, and contradiction removal. Conclusions, further work, as well as potential use, terminate the paper.

Keywords: Logic Program semantics, Stable Models, *Reductio ad Absurdum*.

Tutorials

Proposal of a Tutorial on Answer Set Programming

Stefania Costantini[1]

Dipartimento di Informatica,
Università degli Studi di L'Aquila,
L'Aquila, I-67100 Italy,
stefcost@di.univaq.it
URL http://costantini.di.univaq.it

Abstract

Answer Set Programming (ASP) is an emerging paradigm of logic programming based on the Answer Set (or equivalently Stable Model) semantics: each solution to a problem is represented by an Answer Set (also called a Stable Model) of a deductive database/function-free logic program encoding the problem itself. It is clearly related to deductive databases and knowledge bases, where the occurrence of several answer sets indicates the presence of uncertain or incomplete knowledge, and each answer set represents a possible plausible instance of the database/knowledge base.

Recent work demonstrates that Answer Set Programming is a suitable paradigm for defining and implementing data integration systems. In particular, the author of this proposal has defined a formalization in answer set programming, and a working inference engine, for the Global-as-View approach. The reason why answer set programming is well suited for representing mapping s between data models is exactly that the query answering problem can be coped with also in the p resence of incomplete/ambiguous/ inconsistent data sources: this by means of the advanced reasoning capabilities of computational logic, and by means of the possibility of making different plausible answers to queries explicit, as different answer set.

The tutorial will introduce concepts and notions of computational logic, will describe the DATALOG¬ language and the answer set semantics, and will outline a comparison with traditional logic programming. However, the level of the description will be accessible to the non-expert, by providing few formal details intermixed with several intuitive examples. Some hints and references will be proposed for those who may wish to go into deeper detail.

Il Problema del Protein Folding e i Relativi Approcci Basati su Programmazione con Vincoli

Agostino Dovier
Università degli Studi di Udine

Sommario

Le proteine sono presenti massivamente negli organismi viventi. Strutturalmente una proteina può essere considerata una catena costituita da elementi più semplici, detti aminoacidi. Gli aminoacidi possono essere di 20 tipi, mentre la lunghezza tipica di una proteina va da poche decine a diverse centinaia. Ogni proteina assume una determinata forma spaziale che ne caratterizza la funzione biologica.

Oggigiorno è possibile scoprire la sequenza degli amminoacidi di una proteina, cosi come è possibile generare in laboratorio determinate sequenze di aminoacidi. Non esistono tuttavia ancora dei tools per predire la forma spaziale data la sequenza degli aminoacidi ne tantomeno delle tecniche di laboratorio che permettano di "fotografare" la forma spaziale di proteine esistenti in tempi ragionevoli.

In questo tutorial si mostra come il constraint programming sia adatto ad affrontare il problema della predizione della forma spaziale di una proteina, la cui risoluzione ha ricadute immediate in medicina e nelle biotecnologie in generale.

Introduzione ai Sistemi Multi-Agente basati su Logica Computazionale

Paolo Torroni
DEIS Bologna

Sommario

Negli ultimi anni c'è stata una notevole crescita di interesse verso un paradigma computazionale noto sotto il nome di "sistemi multi-agente". Esso è l'oggetto di studio di una nuova area di ricerca che riunisce ed integra certi aspetti dell'Intelligenza Artificiale con altri tipici dei Sistemi Distribuiti, con l'idea di sviluppare modelli e architetture per agenti intelligenti.

Questo breve tutorial vuole essere un'introduzione ai sistemi multi-agente intelligenti, dal punto di vista dei modelli formali, e del ruolo importante che ha avuto e sta avendo la logica computazionale nell'affrontare problematiche di vario tipo, dalla rappresentazione della conoscenza ai meccanismi di ragionamento, dalla semantica delle interazioni tra gli agenti ai modelli operazionali.

Contributed Talks

Abduction with Hypotheses Confirmation

Marco Alberti[1], Marco Gavanelli[1], Evelina Lamma[1],
Paola Mello[2], and Paolo Torroni[2]

[1] DI - University of Ferrara - Via Saragat, 1 - 44100 Ferrara, Italy.
{malberti|m gavanelli|elamma}@ing.unife.it
[2] DEIS - University of Bologna - Viale Risorgimento, 2 - 40136 Bologna, Italy.
{pmello|ptorroni}@deis.unibo.it

Abstract. Abduction can be seen as the formal inference corresponding to human hypothesis making. It typically has the purpose of explaining some given observation. In classical abduction, hypotheses could be made on events that may have occurred in the past. In general, abductive reasoning can be used to generate hypotheses about events possibly occurring in the future (forecasting), or may suggest further investigations that will confirm or disconfirm the hypotheses made in a previous step (as in scientific reasoning). We propose an operational framework based on Abductive Logic Programming, which extends existing frameworks in many respects, including accommodating dynamic observations and hypothesis confirmation.

1 Introduction

Often, reasoning paradigms in artificial intelligence mimic human reasoning, providing a formalization and a better understanding of the human basic inferences. Abductive reasoning can be seen as a formalization, in computational logics, of hypotheses making. In order to explain observations, we hypothesize that some (unknown) events have happened, or that some (not directly measurable) properties hold true. The hypothesized facts are then assumed as true, unless they are disconfirmed in the following.

Hypothesis making is particularly important in scientific reasoning: scientists will hypothesize properties about nature, which explain some observations; in subsequent work, they will try to prove (if possible), or at least to confirm the hypotheses. This process leads often to generating new alternative sets of hypotheses. Starting from hypotheses on the current situation, scientists try to foresee their possible consequences; this provides new hypotheses on the future behavior that will be confirmed or disconfirmed by the actual events.

A typical application of abductive reasoning is *diagnosis*. Starting from the observation of symptoms, physicians hypothesize in general possible alternative diseases that may have caused them. Following an iterative process, they will try to support their hypotheses, by prescribing further exams, of which they foresee the possible alternative results. They will then drop the hypotheses disconfirmed by such results, and take as most faithful those supported by them. New findings, such as results of exams or new symptoms, may help generating new hypotheses.

We can then describe this kind of hypothetical reasoning process as composed by three main elements: classically, *explaining observations*, by assuming possible

causes of the observed effects; but also, *adapting* such assumptions to upcoming events, such as new symptoms occurring, and *foreseeing* the occurrence of new events, which may or may not occur indeed.

In Abductive Logic Programming, many formalisms have been proposed [1–6], along with proof procedures able to provide, given a knowledge base and some observation, possible sets of hypotheses that explain the observation. Integrity Constraints are used to drive the process of hypothesis generation, to make such sets consistent, and possibly to suggest new hypotheses. Most frameworks focus on one aspect of abductive reasoning: assumption making, based on a static knowledge and on some observation.

In this work, we extend the concepts of abduction and abductive proof procedures in two main directions.

First, we cater for the dynamic acquisition of new facts (events), which possibly have an impact on the abductive reasoning process, and for confirmation (or disconfirmation) of hypotheses based on such events. We propose a language, able to state required properties of the events supporting the hypotheses: for instance, we could say that, given some combination of hypotheses and facts, we make the hypothesis that some new events will occur. We call this kind of hypothesis *expectation*. For this purpose, we express expectations as abducible literals. Expectations can be "positive (to be confirmed by certain events occurring), or "negative" (to be confirmed by certain events not occurring).

Second, in our framework, we need to be able to state that some event is expected to happen *within some time interval*: if the event does actually happen within it, the hypothesis is confirmed, it is disconfirmed otherwise. In doing so, we need to introduce *variables* (e.g. to model time), and to state *constraints* on variables occurring in abducible atoms. Moreover, possible expectation could be involving universal quantification: this typically happens with negative expectations ("The patient is expected *not* to show symptom Q *at all times*"). For this reason, we also need to cater for abducibles possibly containing universally quantified variables.

To summarize, the main new features of the present work with respect to classical ALP frameworks are:

- dynamic update of the knowledge base to cater for new events, whose occurrence interacts with the abductive reasoning process itself;
- confirmation and disconfirmation of hypotheses, by matching expectations and actual events;
- hypotheses with universally quantified variables;
- constraints à la Constraint Logic Programming [7].

We achieve these features by defining syntax, declarative and operational semantics of an abductive framework, based on an extension of the IFF proof procedure [4], called SCIFF[8].[3] Being the SCIFF an extension of an existing abductive

[3] Historically, the name SCIFF is due to the fact that this framework has been firstly applied to modelling protocol in agent Societies, and that is also deals with CLP Constraints.

framework, it also caters for classic abductive logic programming (static knowledge, no notion of confirmation by events). However, due to space limitations, in this work we only focus on the original new parts.

The SCIFF has been implemented using *Constraint Handling Rules* [9] and integrated in a Java-based system for hypothetical reasoning [10].

In the following Sect. 2 we introduce our framework's knowledge representation. In Sect. 3 and 4 we provide declarative and operational semantics, and we show a soundness result. In Sect. 5 we show an example of the functioning of the SCIFF in a multi-agent setting. Before concluding, we discuss about related work in Sect. 6.

Additional details about the syntax of data structures used by the SCIFF and allowedness criteria used to prove soundness are given in [11].

2 Knowledge Representation

In this section we show the knowledge representation of the abstract abductive framework of the SCIFF.

As typical abductive frameworks [12], ours is representated by a 3-tuple $\langle DKB, \mathcal{IC}, A \rangle$ where:

- DKB is the (dynamic) knowledge base, union of KB (a logic program, possibly containing abducibles in the body of the clauses, which does not change during the computation) and **HAP** (the dynamic part, as explained below);
- $\mathcal{A}$ is the set of abducible predicates (in our case, **E**, **EN** and their explicit negations ¬**E** and ¬**EN**); and
- $\mathcal{IC}$ is the set of *Integrity Constraints*.

The set **HAP** is composed of ground facts (defining the **H** predicate), of the form

$$\mathbf{H}(\textit{Description}[, \textit{Time}])$$

where *Description* is a ground term representing the event that happened and *Time* is an integer number representing the time at which the event happened.

The history **HAP** dynamically grows during the computation, as new events happen; we do not model here the source of such events, but it can be imagined as a queue. We assume that events arrive in the same temporal order in which they happen.

The evolution of the history **HAP** reflects in the evolution of the abductive framework, of which **HAP** is part. We can formalize the evolution as a sequence $DALP$ of frameworks, and denote with $DALP_{\mathbf{HAP}}$ the specific instance associated to a specific history **HAP**.

An instance $DALP_{\mathbf{HAP}}$ of an abductive framework is queried with a *goal* $\mathcal{G}$. The goal may contain both predicates defined in KB and abducibles.

The computation of an instance of an abductive framework will produce, by abduction, a set **EXP** of *expectations*, which represent events that are expected to (but might not) happen (*positive* expectations, of the form $\mathbf{E}(\textit{Description}[, \textit{Time}])$)

and events that are expected *not* to (but might) happen (*negative* expectations, of the form $\mathbf{EN}(\mathit{Description}[, \mathit{Time}])$). Typically, expectations will contain variables, over which CLP [7] constraints can be imposed.

The full syntax of our language is reported in [11].

We conclude this section with a simple example in the medical domain, where abduction is used to diagnose diseases starting from symptom observation. The aim of this example is to show the two main improvements of the SCIFF with respect to previous work: the dynamic acquisition of new facts, and the confirmation of hypotheses by events.

Example 1. Let us consider a symptom s, which can be explained by abducing one of three types of diseases, of which the first and the third are incompatible, and the second is accompanied by a condition (the patient's temperature is expected to increase):

$$\begin{array}{l} symptom(s) :- \mathbf{E}(disease(d_1)), \mathbf{EN}(disease(d_3)). \\ symptom(s) :- \mathbf{E}(disease(d_2)), \mathbf{E}(temperature(high)). \\ symptom(s) :- \mathbf{E}(disease(d_3)), \mathbf{EN}(disease(d_1)). \end{array}$$

The set $\mathcal{IC}$ of integrity constraints expresses what is expected or *should* happen or not, given some happened events and/or some abduced hypotheses. They are in the form of implications, and can involve both literals defined in the KB, and expectations and events in **EXP** and **HAP**. For example, an IC in $\mathcal{IC}$ could state that if the result of some exam r is positive, then we can hypothesize that the patient is not affected by disease d_1:

$$\mathbf{H}(result(r, positive)) \rightarrow \mathbf{EN}(disease(d_1))$$

If $\mathbf{H}(result(r, positive))$ actually happens, the integrity constraint would make us abduce $\mathbf{EN}(disease(d_1))$, which would rule out, in our framework, the possibility to abduce $\mathbf{E}(disease(d_1))$. We see how the dynamic occurrence of new events can drive the generation and selection of abductive explanations of goals. Let us now assume that the patient, at some point, shows the symptom $temperature(low)$. The following constraint can be used to express this fact to be inconsistent with an expectation about his temperature increasing:

$$\mathbf{E}(temperature(high)) \rightarrow \mathbf{EN}(temperature(low))$$

If the diagnosis $\mathbf{E}(disease(d_2)), \mathbf{E}(temperature(high))$ is chosen for s, this IC would have as a consequence the generation of the expectation $\mathbf{EN}(temperature(low))$, which would be frustrated by the fact $\mathbf{H}(temperature(low))$. The only possible explanation for s thus remains $\mathbf{E}(disease(d_3)), \mathbf{EN}(disease(d_1))$. We see by this example how the hypotheses can be disconfirmed by events.

The abductive system will usually have a goal, which typically is an observation for which we are searching for explanations; for example, a conjunction of *symptom* atoms.

3 Declarative semantics

In the previous section, we have defined an instance $DALP_{\mathbf{HAP}}$ of an abductive framework as a tuple $\langle DKB, \mathcal{A}, \mathcal{IC}, \rangle$, where $DKB = KB \cup \mathbf{HAP}$. In this section, we propose an abductive interpretation for $DALP_{\mathbf{HAP}}$, depending on the events in the history **HAP**. We adopt a three-valued logic, where literals of kind $\mathbf{H}()$ or $\neg\mathbf{H}()$ can be interpreted as true, false or unknown: when reasoning about future events, it is not possible to state their happening or non-happening.

Throughout this section, as usual when defining declarative semantics, we always consider the ground version of the knowledge base and integrity constraints, and consider CLP-like constraints as defined predicates.

Since an instance $DALP_{\mathbf{HAP}}$ is an abductive logic program, an abductive explanation should entail the goal and satisfy the integrity constraints:

$$KB \cup \mathbf{HAP} \cup \mathbf{EXP} \models G \tag{1}$$

$$KB \cup \mathbf{HAP} \cup \mathbf{EXP} \models IC \tag{2}$$

where, as in the IFF proof procedure [4], the symbol $\models$ stands for three valued completion semantics. Notice, however, that we do not complete the set **HAP**, as it would imply that events no events will happen in the future. In other words, the closed world assumption cannot hold for future events.

Among the sets of expectations computed as abductive explanations for an instance $DALP_{\mathbf{HAP}}$, we select the ones that are consistent with respect to the expected events (i.e., we do not want the same event to be both expected to happen and expected not to happen) and to negation:

Definition 1. *A set of expectations* **EXP** *is* E-consistent *if and only if for each (ground) term p:* $\{\mathbf{E}(p), \mathbf{EN}(p)\} \not\subseteq \mathbf{EXP}$.

A set of expectations **EXP** *is* ¬-consistent *if and only if for each (ground) term p:* $\{\mathbf{E}(p), \neg\mathbf{E}(p)\} \not\subseteq \mathbf{EXP}$ *and* $\{\mathbf{EN}(p), \neg\mathbf{EN}(p)\} \not\subseteq \mathbf{EXP}$.[4]

Finally, we require that the set of expectations computed as an abductive explanation for an instance $DALP_{\mathbf{HAP}}$ be *confirmed*:

Definition 2. Confirmation. *Given a history* **HAP***, a set of expectations* **EXP** *is* confirmed *if and only if for each (ground) term p:*

$$\mathbf{HAP} \cup Comp(\mathbf{EXP}) \cup \{\mathbf{E}(p) \rightarrow \mathbf{H}(p)\} \cup \{\mathbf{EN}(p) \rightarrow \neg\mathbf{H}(p)\} \cup CET \not\models false \tag{3}$$

If Eq. 3 does not hold, the set of expectations is called disconfirmed*.*

Definition 2 requires that each negative expectation in **EXP** have no corresponding happened event; if there is one, **EXP** is disconfirmed. Disconfirmation of a positive expectation, instead, can only occur if some deadline on the expectation (expressed by CLP constraints on the time variable) is missed; otherwise, an event confirming the expectation may always occur in the future.

[4] For abducibles, we adopt the same viewpoint as in ACLP [5]: for each abducible predicate A, we have also the abducible predicate $\neg A$ for the negation of A together with the integrity constraint $(\forall X)\neg A(X), A(X) \rightarrow \bot$.

4 Operational Semantics

Our framework's IC are very much related to the integrity constraints of the IFF proof procedure [4]. This leads to the idea of using an extension of the IFF proof procedure for generating expectations, and check for their confirmation.

In particular, the additional features that we need are the following: (*i*) accept new events as they happen, (*ii*) produce a (disjunction of) set of expectations, (*iii*) detect confirmation of expectations, (*iv*) detect disconfirmation as soon as possible.

The proof procedure that we are about to present is called SCIFF. Following Fung and Kowalski's approach [4], we describe the SCIFF as a transition system. Due to space limitations, we will only focus here on the new transitions, while the reader can refer to [4] for the basic IFF transitions.

4.1 Data Structures

The SCIFF proof procedure is based on a transition system. Each state is defined by the tuple $T \equiv \langle R, CS, PSIC, \mathbf{EXP}, \mathbf{HAP}, \mathbf{CONF}, \mathbf{DISC} \rangle$, where R is the resolvent, CS is the constraint store, $PSIC$ is the set of partially solved integrity constraints, **EXP** is the set of (pending) expectations, **HAP** is the history of happened events, **CONF** is a set of confirmed hypotheses, **DISC** is a set of disconfirmed expectations.

Variable quantification In the IFF proof procedure, all the variables that occur in the resolvent or in abduced literals are existentially quantified, while the others (appearing only in implications) are universally quantified. Our proof procedure has to deal with universally quantified variables in the abducibles and in the resolvent. In the IFF proof procedure, variables in an implication are existentially quantified if they also appear in an abducible or in the resolvent. In our language, we can have existentially quantified variables in the integrity constraints even if they do not occur elsewhere (see [11]).

For all these reasons, in the operational semantic specification we leave the variable quantification explicit. Moreover, we need to distinguish between variables that appear in abduced literals (or in the resolvent) and variables occurring only in integrity constraints. The scope of the variables that occur only in an implication is the implication itself; the scope of the other variables is the whole tuple.

Initial Node and Success A derivation D is a sequence of nodes

$$T_0 \rightarrow T_1 \rightarrow \ldots \rightarrow T_{n-1} \rightarrow T_n.$$

Given a goal G and a set of integrity constraints $\mathcal{IC}$, the first node is:

$$T_0 \equiv \langle \{G\}, \emptyset, \mathcal{IC}, \emptyset, \emptyset, \emptyset, \emptyset \rangle$$

i.e., the resolvent is initially the query ($R_0 = \{G\}$) and the partially solved integrity constraints $PSIC$ is the set of integrity constraints ($PSIC_0 = \mathcal{IC}$).

The other nodes $T_j, j > 0$, are obtained by applying the transitions defined in the next section, until no transition can be applied anymore (quiescence). Every arc in a derivation is labelled with the name of a transition.

Definition 3. *Starting with an instance $DALP_{\mathbf{HAP}^i}$ there exists a* successful derivation *for a goal G iff the proof tree with root node $\langle \{G\}, \emptyset, \mathcal{IC}, \emptyset, \mathbf{HAP}^i, \emptyset, \emptyset \rangle$ has at least one leaf node $\langle \emptyset, CS, PSIC, \mathbf{EXP}, \mathbf{HAP}^f, \mathbf{CONF}, \emptyset \rangle$ where $\mathbf{HAP}^f \supseteq \mathbf{HAP}^i$ and CS is consistent (i.e., there exists a ground variable assignment such that all the constraints are satisfied). In that case, we write:*

$$DALP_{\mathbf{HAP}^i} \vdash^{\mathbf{HAP}^f}_{\mathbf{EXP} \cup \mathbf{CONF}} G$$

Notice that, coherently with the declarative semantics given earlier, a success node cannot contain disconfirmed hypotheses. However, in some applications, all the alternative sets may contain disconfirmed expectations and the goal could be finding a set of expectations with a minimal set of disconfirmed ones. For this reason, we preferred to map explicitly the set of disconfirmed expectations, instead of generating a simple *fail* node (paving the way for future extensions of the framework). Moreover, classical Logic Programming provides explanations (e.g., variable binding) about why a computation succeeds, but no explanation for failure nodes. In our framework, a failure can be explained by means of alternative sets of disconfirmed expectations.

From a non-failure leaf node N, answers can be extracted in a very similar way to the IFF proof procedure. First, a substitution σ' is computed such that σ' replaces all variables in N that are not universally quantified by ground terms, and σ' satisfies all the constraints in the store CS_N.

Definition 4. *Let $\sigma = \sigma'|_{vars(G)}$ be the restriction of σ' to the variables occurring in the initial goal G. Let $\Delta = [\mathbf{CONF}_N \cup \mathbf{EXP}_N)]\sigma'$. The pair (Δ, σ) is the* abductive answer *obtained from the node N.*

The $\mathcal{S}$CIFF proof procedure performs some inferences based on the semantics of time, under the temporal order assumption (see Sect. 2).

Consistency In order to ensure **E**-consistency of the set of expectations, we require that the set $\mathcal{IC}$ of integrity constraints always contain the following:

$$\mathbf{E}(T) \land \mathbf{EN}(T) \rightarrow \bot$$

while for $\neg$-consistency, we impose the following integrity constraints:

$$\mathbf{E}(T) \land \neg\mathbf{E}(T) \rightarrow \bot \qquad \mathbf{EN}(T) \land \neg\mathbf{EN}(T) \rightarrow \bot$$

4.2 Transitions

The transitions are those of the IFF proof procedure, enlarged with those of CLP [7], and with specific transitions accommodating the concepts of confirmation of hypotheses, and dynamically growing history.

In this section, the letter k will indicate the level of a node; each transition will generate one or more nodes from level k to $k+1$. We will not explicitly report the new state for items that do not change; e.g., if a transition generates a new node from the node

$$T_k \equiv \langle R_k, CS_k, PSIC_k, \mathbf{EXP}_k, \mathbf{HAP}_k, \mathbf{CONF}_k, \mathbf{DISC}_k \rangle$$

and we do not explicitly state the value of R_{k+1}, it means that $R_{k+1} = R_k$.

IFF-like transitions The $\mathcal{S}$CIFF proof procedure contains transitions borrowed from the IFF proof procedure, namely *Unfolding*, *Propagation*, *Splitting*, *Case Analysis*, *Factoring*, *Equivalence Rewriting* and *Logical Equivalence*. They have been extended to cope with abducibles containing universally quantified variables and with CLP constraints. We omit them here for lack of space; the basic transitions were proposed by Fung and Kowalski [4], while the extended ones can be found in a technical report [13].

Dynamically growing history The happening of events is dealt with by a transition *Happening*. This transition takes an event $\mathbf{H}(Event)$ from the external queue and puts it in the history **HAP**; the transition *Happening* is applicable only if an *Event* such that $\mathbf{H}(Event) \notin \mathbf{HAP}$ is in the external queue.

Formally, from a node N_k transition *Happening* produces a single successor

$$\mathbf{HAP}_{k+1} = \mathbf{HAP}_k \cup \{\mathbf{H}(Event)\}.$$

Note that transition *Happening* should be applied to all the non-failure nodes (in the frontier).

Confirmation and Disconfirmation

Disconfirmation **EN** Given a node N with the following situation:

$$\mathbf{EXP}_k = \mathbf{EXP'} \cup \{\mathbf{EN}(E_1)\} \qquad \mathbf{HAP}_k = \mathbf{HAP'} \cup \{\mathbf{H}(E_2)\}$$

Disconfirmation **EN** produces two nodes N^1 and N^2, as follows:

N^1	N^2
$\mathbf{EXP}^1_{k+1} = \mathbf{EXP'}$	$\mathbf{EXP}^2_{k+1} = \mathbf{EXP}_k$
$\mathbf{DISC}^1_{k+1} = \mathbf{DISC}_k \cup \{\mathbf{EN}(E_1)\}$	$\mathbf{DISC}^2_{k+1} = \mathbf{DISC}_k$
$CS^1_{k+1} = CS_k \cup \{E_1 = E_2\}$	$CS^2_{k+1} = CS_k \cup \{E_1 \neq E_2\}$

Remember that node N^1 is a failure node, as in success nodes $\mathbf{DISC} = \emptyset$ (see Sect. 3).

Example 2. Suppose that $\mathbf{HAP}_k = \{\mathbf{H}(p(1,2))\}$ and $\exists X \forall Y \mathbf{EXP}_k = \{\mathbf{EN}(p(X,Y))\}$. *Disconfirmation* **EN** will produce the two following nodes:

$\exists X \forall Y \mathbf{EXP}_k = \{\mathbf{EN}(p(X,Y))\}$ $\mathbf{HAP}_k = \{\mathbf{H}(p(1,2))\}$

$CS^1_{k+1} = \{X = 1 \wedge Y = 2\}$ $CS^2_{k+1} = \{X \neq 1 \vee Y \neq 2\}$
$\mathbf{DISC}^1_{k+1} = \{\mathbf{EN}(p(1,2))\}$
$CS_{k+2} = \{X \neq 1\}$

where the last simplification in the right branch is due to the rules of the constraint solver (see Section CLP).

Confirmation **E** Starting from a node N as follows:

$$\mathbf{EXP}_k = \mathbf{EXP}' \cup \{\mathbf{E}(E_1)\}, \qquad \mathbf{HAP}_k = \mathbf{HAP}' \cup \{\mathbf{H}(E_2)\}$$

Confirmation **E** builds two nodes, N^1 and N^2; in node N^1 we assume that the expectation and the happened event unify, and in node N^2 we hypothesize that the two do not unify:

N^1		N^2
$\mathbf{EXP}^1_{k+1} = \mathbf{EXP}'$		$\mathbf{EXP}^2_{k+1} = \mathbf{EXP}_k$
$\mathbf{CONF}^1_{k+1} = \mathbf{CONF}_k \cup \{\mathbf{E}(E_1)\}$		$\mathbf{CONF}^2_{k+1} = \mathbf{CONF}_k$
$CS^1_{k+1} = CS_k \cup \{E_1 = E_2\}$		$CS^2_{k+1} = CS_k \cup \{E_1 \neq E_2\}$

In this case, N^2 is not a failure node, as $\mathbf{E}(E_1)$ might be confirmed by other events.

Disconfirmation **E** In order to check the disconfirmation of a positive expectation **E**, we need to assume that there will never be a matching event in the external queue. We can, e.g., exploit the semantics of *time*. If we make the hypothesis of temporal ordering (Sect. 2), we can infer that an expected event for which the deadline is passed, is disconfirmed.

Given a node:

- $\mathbf{EXP}_k = \{\mathbf{E}(X,T)\} \cup \mathbf{EXP}'$
- $\mathbf{HAP}_k = \{\mathbf{H}(Y,T_c)\} \cup \mathbf{HAP}'$
- $\forall E_2, T_2 : \mathbf{H}(E_2,T_2) \in \mathbf{HAP}_k$, $CS_k \cup \{(E_2,T_2) = (X,T)\} \models \bot$
- $CS_k \models T < T_c$

transition Disconfirmation **E** is applicable and creates the following node:

- $\mathbf{EXP}_{k+1} = \mathbf{EXP}'$
- $\mathbf{DISC}_{k+1} = \mathbf{DISC}_k \cup \{\mathbf{E}(X,T)\}$.

Operationally, one can often avoid checking that (X,T) does not unify with every event in the history by choosing a preferred order of application of the transitions. By applying Disconfirmation **E** only if no other transition is applicable, the check can be safely avoided, as the test of confirmation is already performed by *Confirmation* **E**.

Notice that this transition infers the current time from happened event; i.e., it infers that the current time cannot be less than the time of a happened event.

Symmetrically to Disconfirmation **E**, we also have a transition *Confirmation* **EN**, which we do not report here for lack of space; the interested reader is referred to [13].

Note that the entailment of constraints from a constraint store is, in general, not easy to verify. In the particular case of $CS_k \models T < T_c$, however, we have that the constraint $T < T_c$ is unary (T_c is always ground), thus a CLP for finite domains solver CLP(FD) is able to verify the entailment very easily if the store contains only unary constraints (it is enough to check the maximum value in the domain of T). Moreover, even if the store contains non-unary constraints (thus the solver performs, in general, incomplete propagation), the transition will not compromise the soundness and completeness of the proof procedure. If the solver performs a powerful propagation (including pruning, in CLP(FD)), the disconfirmation will be early detected, otherwise, it will be detected later on.

CLP Here we adopt the same transitions as in CLP [7]. Therefore, the symbols $=$ and $\neq$ are in the constraint language. Note that a constraint solver works on a constraint domain which has an associated interpretation. In addition, the constraint solver should handle the constraints among terms derived from the unification. Therefore, beside the specific constraint propagation on the constraint domain, we need further inference rules for coping with the unification. For space limitations, we cannot show them here: but they can be found in [11].

4.3 Soundness

The following proposition relates the operational notion of successful derivation with the corresponding declarative notion of goal provability.

Proposition 1. *Given an instance $ALP_{\mathbf{HAP}^i}$ of an ALP program and a ground goal G, if $DALP_{\mathbf{HAP}^i} \mid\!\sim^{\mathbf{HAP}^f}_{\mathbf{EXP} \cup \mathbf{CONF}} G$ then $DALP_{\mathbf{HAP}^f} \mid\!\approx_{\mathbf{EXP} \cup \mathbf{CONF}} G$.*

This property has been proven for some notable classes of ALP programs. In particular, a proof of soundness can be found in [13] for *allowed* ALPs (for a definition of allowedness see [11]). The proof is based on a correspondence drawn between the $\mathcal{S}$CIFF and IFF transitions, and exploits the soundness results of the IFF proof procedure [4].

5 Using the $\mathcal{S}$CIFF for agent compliance verification

Abduction has been used for various applications, and many of them (e.g. diagnosis) could benefit from an extension featuring hypotheses confirmation, such as the one depicted in this paper. We have applied the language to a multi-agent setting, in the context of the SOCS project [14].

In order to combine autonomous agents and have them operate in a coordinated fashion, protocols are often defined. Protocols show, in a way, the ideal behaviour of agents. But, since agents are often assumed to be autonomous, and societies open and heterogeneous, agent compliance to protocols is rarely a reasonable assumption to make. Agents may violate the protocols due to malicious

intentions, to wrong design or, for instance, to failure to keep the pace with tight deadlines.

With our language, protocols can be easily formalized, and the $\mathcal{S}$CIFF proof procedure can be used then to check whether the agents comply to protocols. For instance, let us consider the (very simple) *query-ref* protocol: an agent requests a piece of information to another agent, which, by a given deadline, should either provide it or refuse it, but not both. The protocol can be expressed by the following three integrity constraints (where D represents a dialogue identifier):

$\mathbf{IC_1}$: $\mathbf{H}(tell(A, B, query\text{-}ref(Info), D), T) \wedge deadline(T_d) \rightarrow$
$\mathbf{E}(tell(B, A, inform(Info, Answer), D), T_1) \wedge T_1 \leq T + T_d \vee$
$\mathbf{E}(tell(B, A, refuse(Info), D), T_1) \wedge T_1 \leq T + T_d$

$\mathbf{IC_2}$: $\mathbf{H}(tell(A, B, inform(Info, Answer), D), T) \rightarrow$
$\mathbf{EN}(tell(A, B, refuse(Info), D), T_1) \wedge T_1 \geq T$

$\mathbf{IC_3}$: $\mathbf{H}(tell(A, B, refuse(Info), D), T) \rightarrow$
$\mathbf{EN}(tell(A, B, inform(Info, Answer), D), T_1) \wedge T_1 \geq T$

IC_1 expresses that an agent that receives a *query-ref* must reply with either an *inform* or a *refuse* by T_d time units; IC_2 and IC_3 state that an agent that performs an *inform* cannot perform a *refuse* later, and *vice-versa*. Predicate *deadline*/1 is defined in the KB: in this example, let it be defined by the one fact *deadline*(*10*).

Let us suppose that the following events happen:

$\mathbf{H_1}$: $\mathbf{H}(tell(alice, bob, query\text{-}ref(what\text{-}time), d_0), 10)$

$\mathbf{H_2}$: $\mathbf{H}(tell(bob, alice, refuse(what\text{-}time), d_0), 15)$

and consider how $\mathcal{S}$CIFF verifies that such an history is compliant to the interaction protocol.

The $\mathbf{H_1}$ event, by propagation of $\mathbf{IC_1}$ (where, by unfolding, T_d has been bound to 10), will cause a disjunction of two expectations to be generated, which will split the proof tree into two branches:

1. In the first branch, $\mathbf{EXP} = \{\mathbf{E}(tell(bob, alice, inform(what\text{-}time, Answer), d_0), T_1)\}$ and $\mathbf{CS} = \{T_1 \leq 20\}$. When $\mathbf{H_2}$ happens, by propagation of $\mathbf{IC_2}$, $\mathbf{EXP} = \{\mathbf{E}(tell(bob, alice, inform(what\text{-}time, Answer), d_0), T_1),$
$\mathbf{EN}(tell(bob, alice, inform(what\text{-}time, Answer), d_0), T_2)\}$, with $\mathbf{CS} = \{T_1 \leq 20, T_2 \geq 15\}$. Then, by enforcing E-consistency, the domain of T_1 is reduced: $\mathbf{CS} = \{T_1 \leq 14, T_2 \geq 15\}$. Now, *Disconfirmation*-$\mathbf{E}$ can be applied:
$\mathbf{E}(tell(bob, alice, inform(what\text{-}time, Answer), d_0), T_1)$ is moved from **EXP** to **DISC**; this means that this branch cannot be successful.
2. In the second branch, $\mathbf{EXP} = \{\mathbf{E}(tell(bob, alice, refuse(what\text{-}time), d_0), T_1)\}$ and $\mathbf{CS} = \{T_1 \leq 20\}$. By propagation of $\mathbf{IC_2}$, after $\mathbf{H_2}$,
$\mathbf{EXP} = \{\mathbf{E}(tell(bob, alice, refuse(what\text{-}time), d_0), T_1),$
$\mathbf{EN}(tell(bob, alice, inform(what\text{-}time, Answer), d_0), T_2)\}$, with $\mathbf{CS} = \{T_1 \leq 20, T_2 \geq 15\}$. By *Confirmation* $\mathbf{E}$, $\mathbf{H_2}$ also causes
$\mathbf{E}(tell(bob, alice, refuse(what\text{-}time), d_0), T_1)$ to be moved from **EXP** to **CONF**, with $T_1 = 15$. If no more events happen, this branch is successful.

Through this simple example, we showed how a protocol can be easily cast in our model. More rules can be easily added, to accomplish more complex protocols. In other work, we have shown the application of this formalism to a range of protocols [15, 16]. The use of expectations generated by the $\mathcal{S}$CIFF could be

manifold: by associating Confirmation/Disconfirmation with a notion of Fulfillment/Violation, we can verify at run-time the compliance of agents to protocols. Moreover, expectations, if made public, could be used by agents planning their activities, helping their choices if they aim at exhibiting a compliant behaviour.

6 Related Work

This work is mostly related to the IFF proof procedure [4], which it extends in several directions, as stated in the introduction.

Other proof procedures deal with constraints; in particular we mention ACLP [5] and the $\mathcal{A}$-system [6], which are deeply focussed on efficiency issues. Both of these proof procedures use integrity constraints in the form of denials (e.g., $A, B, C \rightarrow \perp$), instead of forward rules as the IFF (and SCIFF). Both of these proof procedures only abduce existentially quantified atoms, and do not consider quantifier restrictions, which make the SCIFF in this sense more expressive.

Some conspicuous work has been done with the integration of the IFF proof procedure with constraints [17]; however the integration is more focussed on a theoretical uniform view of abducibles and constraints than to an implementation of a proof procedure with constraints.

In [18], Endriss et al. present an implementation of an abductive proof procedure that extends IFF [4] in two ways: by dealing with constraint predicates and with non-allowed abductive logic programs. The cited work, however, does not deal with confirmation and disconfirmation of hypotheses and universally quantified variables in abducibles (**EN**), as ours does. The two works also differ in their implementation: Endriss et al.'s is a metainterpreter which exploits a built-in constraint solver, whereas we implement the proof transitions and variable unification by means of CHR and attributed variables. Both works have been conducted in the context of the SOCS project [14]: the main application of Endriss et al.'s is the implementation of the internal agent reasoning, while ours is the compliance check of the observable (external) agent behaviour.

Abdual [19] is a systems for performing abduction from extended logic programs adopting the well-founded semantics. It handles only ground programs. It relies on tabled evaluation and is inspired to SLG resolution [20].

Many other abductive proof procedures have been proposed in the past; the interested reader can refer to the exhaustive survey by Kakas et al. [12].

In [21], Sergot proposed a general framework, called *query-the-user*, in which some of the predicates are labelled as "askable"; the truth of askable atoms can be asked to the user. The framework provides, thus, the possibility of gathering new information during the computation. Our **E** predicates may in a sense be seen as asking information, while **H** atoms may be considered as new information provided during search. However, as we have shown in the paper, **E** atoms may also mean *expected* behavior, and the SCIFF can cope with abducibles containing universally quantified variables.

The concept of hypotheses confirmation has been studied also by Kakas and Evans [22], where hypotheses can be corroborated or refuted by matching them with observable atoms: an explanation fails to be corroborated if some of its

logical consequences are not observed. The authors suggest that their framework could be extended to take into account dynamic events, eventually, queried to the user: *"this form of reasoning might benefit for the use of a query-the-user facility"*.

In a sense, our work can be considered as an extension of these works: it provides the concept of confirmation of hypotheses, as in corroboration, and it provides an operational semantics for dynamically incoming events. Moreover, we extend the work by imposing integrity constraints to better define the feasible combinations of hypotheses, and we let the program abduce non-ground atoms.

The dinamicity of incoming events can be considered as an instance of an Evolving Logic Program [23]. In EvoLP, the knowledge base can be dynamically modified depending both on events that come from the external world and on the result of a previous computation. The SCIFF proof procedure does not generate new events, but only expectations about external events. The focus of the work is more on the expressivity of the generated expectations (which can contain variables universally quantified and constrained) than on the generality of the evolution of the knowledge base. An interesting extension of our work would contain evolution of the whole knowledge bases, not only of the set of happened events.

In Speculative Computation [24, 25] hypotheses are abduced and can be confirmed later on. It is a framework for a multi-agent setting with unreliable communication. When an agent asks a query to another agent, it also abduces its (default) answer; if the real answer arrives within a deadline, the hypothesis is confirmed or disconfirmed; otherwise the computation continues with the default. In our work, expectations can be confirmed by events, but the scope is different. In our work, if a deadline is missed the computation fails, as an hypothesis has been disconfirmed.

Other implementations have been given of abductive proof procedures in Constraint Handling Rules [26, 27]. Our implementation is more adherent to the theoretical operational semantics (in fact, every transition is mapped onto CHR rules) and exploits the uniform understanding of constraints and abducibles noted by Kowalski et al. [17].

Finally, in Section 5 we considered multi-agent systems to show an application of the SCIFF. Some discussion about other formal approaches to protocol verification can be found in [13].

7 Conclusions

In this paper, we proposed an abductive logic programming framework which extends most previous work in several directions. The two main features of this framework are: the possibility to account for new dynamically upcoming facts, and the possibility to have hypotheses confirmed/disconfirmed by following observations and evidence. We proposed a language, and described its declarative and operational semantics. We implemented the proof-procedure for a system verifying the compliance of agents to protocols; the implementation can be downloaded from `http://lia.deis.unibo.it/Research/sciff/` [8].

Acknowledgements

This work has been supported by the European Commission within the SOCS project (IST-2001-32530), funded within the Global Computing programme and by the MIUR COFIN 2003 projects *La Gestione e la negoziazione automatica dei diritti sulle opere dell'ingegno digitali: aspetti giuridici e informatici* and *Sviluppo e verifica di sistemi multiagente basati sulla logica.*

References

1. Poole, D.L.: A logical framework for default reasoning. Artificial Intelligence **36** (1988) 27–47
2. Kakas, A.C., Mancarella, P.: On the relation between Truth Maintenance and Abduction. In Fukumura, T., ed.: Proceedings of PRICAI-90, Nagoya, Japan, Ohmsha Ltd. (1990) 438–443
3. Console, L., Dupré, D.T., Torasso, P.: On the relationship between abduction and deduction. Journal of Logic and Computation **1** (1991) 661–690
4. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming **33** (1997) 151–165
5. Kakas, A.C., Michael, A., Mourlas, C.: ACLP: Abductive Constraint Logic Programming. Journal of Logic Programming **44** (2000) 129–177
6. Kakas, A.C., van Nuffelen, B., Denecker, M.: A-System: Problem solving through abduction. In Nebel, B., ed.: Proceedings of the 17th IJCAI, Seattle, Washington, USA, Morgan Kaufmann Publishers (2001) 591–596
7. Jaffar, J., Maher, M.: Constraint logic programming: a survey. Journal of Logic Programming **19-20** (1994) 503–582
8. : (The SCIFF abductive proof procedure) `http://lia.deis.unibo.it/Research/sciff/`.
9. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming **37** (1998) 95–138
10. Alberti, M., Chesani, F.: The implementation of a system for generation and confirmation of hypotheses. Technical Report CS-2004-2, Dipartimento di Ingegneria, Università degli Studi di Ferrara, Ferrara, Italy (2004) Available at `http://www.ing.unife.it/informatica/tr/CS-2004-02.pdf`.
11. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Abduction with hypotheses confirmation. Technical Report CS-2004-01, Department of Engineering, University of Ferrara, Ferrara, Italy (2004) Available at `/http://www.ing.unife.it/informatica/tr/CS-2004-01.pdf`.
12. Kakas, A.C., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. In Gabbay, D.M., Hogger, C.J., Robinson, J.A., eds.: Handbook of Logic in Artificial Intelligence and Logic Programming. Volume 5., Oxford University Press (1998) 235–324
13. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of interaction protocols: a computational logic approach based on abduction. Technical Report CS-2003-03, DI, University of Ferrara, Italy (2003) `http://www.ing.unife.it/informatica/tr/cs-2003-03.pdf`.
14. : (Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees) `http://lia.deis.unibo.it/Research/SOCS/`.

15. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Modeling interactions using *Social Integrity Constraints*: a resource sharing case study. (In Leite, J.A., Omicini, A., Sterling, L., Torroni, P., eds.: DALT 2003. Melbourne, Victoria. Workshop Notes) 81–96
16. Alberti, M., Daolio, D., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of agent interaction protocols in a logic-based system. In: Proceedings of the 19th ACM SAC. (AIMS Track), Nicosa, Cyprus, ACM Press (2004) to appear.
17. Kowalski, R., Toni, F., Wetzel, G.: Executing suspended logic programs. Fundamenta Informaticae **34** (1998) 203–224 `http://www-lp.doc.ic.ac.uk/UserPages/staff/ft/PAPERS/slp.ps.Z`.
18. Endriss, U., Mancarella, P., Sadri, F., Terreni, G., Toni, F.: Abductive Logic Programming with CIFF. In Bennett, B., ed.: Proceedings of the 11th Workshop on Automated Reasoning, Bridging the Gap between Theory and Practice, University of Leeds (2004) Extended Abstract. To appear.
19. Alferes, J., Pereira, L.M., Swift, T.: Abduction in well-founded semantics and generalized stable models via tabled dual programs. Theory and Practice of Logic Programming (2003)
20. Chen, W., Warren, D.S.: Tabled evaluation with delaying for general logic programs. Journal of the ACM **43** (1996) 20–74
21. Sergot, M.J.: A query-the-user facility of logic programming. In Degano, P., Sandwell, E., eds.: Integrated Interactive Computer Systems, North Holland (1983) 27–41
22. Evans, C., Kakas, A.: Hypotheticodeductive reasoning. In: Proc. International Conference on Fifth Generation Computer Systems, Tokyo (1992) 546–554
23. Alferes, J.J., Brogi, A., Leite, J.A., Pereira, L.M.: Evolving logic programs. In Flesca, S., Greco, S., Leone, N., Ianni, G., eds.: Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02). Volume 2424 of Lecture Notes in Artificial Intelligence., Springer-Verlag (2002) 50–61
24. Satoh, K., Inoue, K., Iwanuma, K., Sakama, C.: Speculative computation by abduction under incomplete communication environments. In: Proceedings of the 4th International Conference on Multi-Agent Systems, Boston, USA, IEEE Press (2000) 263–270
25. Satoh, K., Yamamoto, K.: Speculative computation with multi-agent belief revision. In Castelfranchi, C., Lewis Johnson, W., eds.: Proceedings of AAMAS-2002, Part II, Bologna, Italy, ACM Press (2002) 897–904
26. Abdennadher, S., Christiansen, H.: An experimental CLP platform for integrity constraints and abduction. In Larsen, H., Kacprzyk, J., Zadrozny, S., Andreasen, T., Christiansen, H., eds.: FQAS, Flexible Query Answering Systems. LNCS, Warsaw, Poland, Springer-Verlag (2000) 141–152
27. Gavanelli, M., Lamma, E., Mello, P., Milano, M., Torroni, P.: Interpreting abduction in CLP. In Buccafurri, F., ed.: AGP, Reggio Calabria, Italy (2003) 25–35

Abductive Logic Programming with CIFF: Implementation and Applications

U. Endriss[1], P. Mancarella[2], F. Sadri[1], G. Terreni[2], and F. Toni[1,2]

[1] Department of Computing, Imperial College London
Email: {ue,fs,ft}@doc.ic.ac.uk
[2] Dipartimento di Informatica, Università di Pisa
Email: {paolo,terreni,toni}@di.unipi.it

Abstract. We describe a system implementing a novel extension of Fung and Kowalski's IFF abductive proof procedure which we call CIFF, and its application to realise intelligent agents that can construct (partial or complete) plans and react to changes in the environment. CIFF extends the original IFF procedure in two ways: by dealing with constraint predicates and by dealing with non-allowed abductive logic programs.

1 Introduction

Abduction has long been recognised as a powerful mechanism for hypothetical reasoning in the presence of incomplete knowledge [9, 13]. In this paper, we discuss the implementation and applications of a novel abductive proof procedure, which we call CIFF. This procedure extends the IFF proof procedure of Fung and Kowalski [10] and is described in detail in [8].

A number of abductive proof procedures have been proposed in the literature [12]. Kakas and Mancarella [13] extended the (first ever) abductive proof procedure of Eshghi and Kowalski [9] for normal logic programming to Abductive Logic Programming (ALP). This has been augmented to deal with constraint predicates in [15] and with integrity constraints that behave like condition-action rules in [22], and has been reformulated to improve its complexity in [21]. All these procedures are proved correct wrt. the (partial) stable model semantics. Another "family" of abductive proof procedures extend that of Console et al. [5] and are proved correct wrt. the completion semantics [4, 20]: these are the IFF procedure [10], and its extensions to deal with constraint predicates, treated as abducibles [19], and integrity constraints which behave like condition-action rules [24]; and the SLDNFA procedure [6], and its extensions to deal with constraint predicates [7]. Some of these procedures have been implemented and experimented with in a number of applications, e.g. [15]. Moreover, recently a system combining features of ACLP [15] and SLDNFA [6], and using heuristic search to improve efficiency, has been proposed in the form of the A-system [17].

Our interest in combinations of ALP and constraint reasoning stems from our work on using computational logic-based techniques in the area of multiagent systems and global computing (for instance, to implement an agent's planning

capability) [14, 27]. We have found, however, that our requirements for these applications go beyond available state-of-the-art ALP proof procedures. This consideration has led us to devise the CIFF proof procedure [8]. The novelty of CIFF is twofold: (1) it dynamically deals with non-allowed programs (i.e. programs with problematic quantification patterns that cannot be handled by the original IFF procedure), thus having wider applicability; and (2) it extends IFF by integrating the procedure with a "black box" constraint solver (rather than treating constraint predicates as abducibles as in [19]). Moreover, CIFF inherits from the original IFF procedure its forward reasoning as well as its syntax of integrity constraints (which is more general than that of most other procedures) and its flexible handling of variables, all of which have been listed as advantages of IFF over other proof procedures. In the present paper, we describe the first implementation of CIFF and illustrate one of its potential applications, namely planning, in some detail. Differently from conventional practice in logic programming, we consider *partial* (rather than *complete*) planning as well as plan repair (via reactivity) in a dynamic environment. We have used CIFF in the PROSOCS system [27], a computational logic-based platform for programming intelligent agents, to build the planning and reactivity components of an agent.

The remainder of this paper is structured as follows. Section 2 provides a brief introduction to ALP and reviews the definition of the CIFF proof procedure. This theoretical presentation is complemented by Section 3, where we show how we have implemented the procedure in Prolog. The application of CIFF to planning is discussed in Section 4. Section 5 concludes.

2 Abductive Logic Programming with CIFF

In this section, we briefly review the framework of Abductive Logic Programming (ALP) for knowledge representation and reasoning [12], as well as the CIFF proof procedure for ALP [8]. ALP can be usefully extended with the handling of constraint predicates in the same way as Constraint Logic Programming (CLP) [11] extends logic programming (see e.g. [15, 19]). Throughout this paper, we assume that our language includes a number of constraint predicates.

2.1 Abductive Logic Programming with Constraints

An *abductive logic program* is a triple $\langle P, I, A \rangle$, where P is a normal logic program (with constraints), I is a finite set of sentences in the language of P (called *integrity constraints*), and A is a set of abducible predicates in the language of P, not occurring in the head of any clause of P [12]. A *query* Q is a conjunction of literals. Any variables occurring in Q are implicitly existentially quantified. These variables are also called the *free* variables.

Broadly speaking, given a program $\langle P, I, A \rangle$ with constraint predicates and a query Q, the purpose of abduction is to find a (possibly minimal) set of abducible atoms (namely atoms whose predicate is abducible) Δ which, together with P and the constraint structure over which the constraint predicates are defined [11],

"entail" (an appropriate ground instantiation of) Q, with respect to a suitable notion of "entailment", in such a way that the extension of P with this set "satisfies" I (see [12] for possible notions of integrity constraint satisfaction). The appropriate notion of "entailment" depends on the semantics associated with the logic program P (again, there are several possible choices for such a semantics [12]). In the remainder of this paper we will adopt the *completion semantics* [4, 20] for logic programming, and extend it *à la* CLP to take the constraint structure into account. We represent entailment under such semantics as $\models_{\Re}$. An *abductive answer* to a query Q for a program $\langle P, I, A\rangle$, containing constraint predicates defined over a structure $\Re$, is a pair $\langle \Delta, \sigma\rangle$, where Δ is a set of ground abducible atoms and σ is a substitution for the free variables in Q such that $P \cup \Delta\sigma \models_{\Re} I \wedge Q\sigma$.

2.2 The CIFF Proof Procedure

We now give a brief description of the CIFF procedure [8]. Like Fung and Kowalski [10], we require the theory given as input to be represented in "iff-form" [4, 10], which we can obtain by *selectively completing* P with respect to all predicates except *special* predicates (*true*, *false*, constraint and abducible predicates). That is, an alternative representation of an abductive logic program would be a pair $\langle Th, I\rangle$, where Th is a set of (homogenised) iff-definitions:

$$p(X_1, \ldots, X_k) \Leftrightarrow D_1 \vee \cdots \vee D_n$$

Here, p must not be a special predicate and there can be at most one iff-definition for every predicate symbol. Each of the disjuncts D_i is a conjunction of literals. The variables $X_1, \ldots, X_k$ are implicitly universally quantified with the scope being the entire definition. Any other variable is implicitly existentially quantified, with the scope being the disjunct in which it occurs.

In this paper, the set of integrity constraints I are implications of the form:

$$L_1 \wedge \cdots \wedge L_m \Rightarrow A_1 \vee \cdots \vee A_n$$

Each of the L_i must be a literal; each of the A_i must be an atom. Any variables are implicitly universally quantified with the scope being the entire implication.

In CIFF, the search for abductive answers of queries over a proof tree is initialised with the root of the tree consisting of the integrity constraints in the program and the literals of the query. The procedure then repeatedly manipulates the current node by applying equivalence-preserving *proof rules* to it. The nodes are sets of formulas (the so-called *goals*) which may be atoms, implications, or disjunctions of literals. The implications are either integrity constraints, their residues, or obtained by rewriting negative literals (e.g. *not* p is rewritten as $p \Rightarrow$ *false*.) The proof rules modify nodes by rewriting goals in them, adding new goals to them, or deleting superfluous goals from them. The set of iff-definitions is kept in the background and is only used to *unfold* defined predicates.

Fung and Kowalski [10] require inputs to their proof procedure to meet a number of allowedness conditions (essentially avoiding certain problematic

patterns of quantification) to be able to guarantee its correct operation. These conditions are overly restrictive; IFF could produce correct answers also for some non-allowed inputs. Unfortunately, it is difficult to formulate appropriate allowedness conditions that guarantee a correct execution of the proof procedure without imposing too many unnecessary restrictions. Our proposal, put forward in [8], is to tackle the issue of allowedness *dynamically*, i.e. at runtime, rather than adopting a static and overly strict set of conditions. To this end, CIFF includes a *dynamic allowedness rule* amongst its proof rules, which gets triggered once the procedure encounters, in the current node, formulas it cannot manipulate correctly due to a problematic quantification pattern. When this happens, the node is labelled as *undefined* and will not be selected again. In addition to the dynamic allowedness rule, the main proof rules of CIFF are:

- *Unfolding:* Replace any atomic goal $p(\vec{t})$, for which there is a definition $p(\vec{X}) \Leftrightarrow D_1 \vee \cdots \vee D_n$ in Th, by $(D_1 \vee \cdots \vee D_n)[\vec{X}/\vec{t}]$. There is a similar rule for defined predicates inside implications.
- *Splitting:* Rewrite any node with a disjunctive goal as a disjunction of nodes.
- *Propagation:* Given goals of the form $p(\vec{t}) \wedge A \Rightarrow B$ and $p(\vec{s})$, add the goal $(\vec{t} = \vec{s}) \wedge A \Rightarrow B$.
- *Case analysis for constraints:* Replace any goal of the form $Con \wedge A \Rightarrow B$, where Con is a constraint not containing any universally quantified variables, by $[Con \wedge (A \Rightarrow B)] \vee \overline{Con}$. There is a similar case analysis rule for equalities.
- *Constraint solving:* Replace any node containing an unsatisfiable set of constraints (as atoms) by *false*.

In addition, there are logical simplification rules, rules for rewriting equalities and making substitutions, and rules that reflect the interplay between constraint predicates and the usual equality predicate. For full details we refer to [8].

In a proof tree for a query, a node containing *false* is called a *failure node*. If all leaf nodes are failure nodes, then the search is said to *fail*. A node to which no more proof rules can be applied is called a *final node*. A final node that is not a failure node and which has not been labelled as *undefined* is called a *success node*. If a success node exists, then the search is said to *succeed*. CIFF has been proved *sound* in [8]: it is possible to extract an abductive answer from any success node (*soundness of success*); and if the search fails then there exists no such answer (*soundness of failure*).

3 Implementation of the Proof Procedure

We have implemented the CIFF procedure in Sicstus Prolog [28].[3] Most of the code could very easily be ported to any other Prolog system conforming to standard Edinburgh syntax. A minor exception is the module concerned with constraint solving as it relies on Sicstus' built-in constraint logic programming solver over finite domains (CLPFD) [3]. However, the modularity of our implementation would make it relatively easy to integrate a different constraint solver

[3] The system is available at `http://www.doc.ic.ac.uk/~ue/ciff/`

```
lamp(X) iff [[X=a]].
battery(X,Y) iff [[X=b, Y=c]].
faulty(X) iff [[lamp(X), broken(X)], [power_failure(X), not(backup(X))]].
backup(X) iff [[battery(X,Y), not(empty(Y))]].
```

Table 1. The abductive logic program for the faulty-lamp example of [10]

into the system instead. The only changes required would be an appropriate re-implementation of a handful of simple predicates providing a wrapper around the constraint solver chosen for the current implementation.

This section discusses various aspects of our implementation of the CIFF procedure and explains how to use the system in practice.

3.1 Representation of Abductive Logic Programs

The CIFF procedure is defined over (selectively) completed logic programs, i.e. sets of definitions in iff-form rather than rules (in if-form) and facts. As these definitions can become rather long and difficult to read, our implementation includes a simple module that translates logic programs into completed logic programs which are then used as input to the CIFF procedure. Being able to complete logic programs on the fly also allows us to spread the definition of a particular predicate over different knowledge bases.

The syntax chosen to represent facts and rules of a logic program is that of Prolog, except that negative literals are represented as Prolog terms of the form `not(`A`)`. In addition, we also allow for (arithmetic) constraints as subgoals in the condition of a rule. For the current implementation, admissible constraints are terms such as `T1 #< T2 + 5`. The available constraint predicates are `#=`, `#\=`, `#<`, `#=<`, `#>`, and `#>=`, each of which takes two arguments that may be any arithmetic expressions over variables and integers (using operators such as addition, subtraction, and multiplication, or any other arithmetic operation that the CLPFD module of Sicstus Prolog can handle [3]). Note that for equalities over terms that are not arithmetic terms, the usual equality predicate `=` should be used (e.g. `X = bob`). Iff-definitions are terms of the form A `iff` B, where A is an atom and B is a list of lists of literals (representing a disjunction of conjunctions). Integrity constraints are expressions of the form A `implies` B, where A is a list of literals (representing a conjunction) and B is a list of atoms (representing a disjunction). Table 1 shows an example using our syntax.

3.2 Running the Program

The main predicate of our implementation is called `ciff/4`:

```
ciff( +Defs, +ICs, +Query, -Answer).
```

The first argument is a list of iff-definitions, the second is a list of integrity constraints, and the third is the list of literals in the query. The `Answer` consists of three parts: a list of abducible atoms, a list of restrictions on the answer

substitution, and a list of constraints (the latter two can be used to construct an answer substitution according to the semantics of ALP).

Alternatively, the first two arguments of `ciff/4` may be replaced with the name of a file containing an abductive logic program. An example for such a file is given in Table 1. This is the *faulty-lamp* example discussed, amongst others, by Fung and Kowalski [10]. The syntax is almost self-explanatory (recall that a list of lists represents a disjunction of conjunctions). This program happens to consist only of iff-definitions (there are no integrity constraints). Assuming that the program is stored in a file called `lamp.alp`, we may run the following query:

```
?- ciff( 'lamp.alp', [faulty(X)], Answer).
Answer = [broken(a)]:[X/a]:[] ? ;
Answer = [empty(c),power_failure(b)]:[X/b]:[] ? ;
Answer = [power_failure(X)]:[not(X/b)]:[] ? ; No
```

Here the user has enforced backtracking, so all three answer are being reported by the system. Note that the third (empty) list in each of the answers would be used to store the associated constraints (of which there are none in this example). For details on how to interpret the above answers, we refer to [10].

3.3 Implementation of the Proof Rules

We are now going to turn our attention to the actual implementation of the proof procedure and explain some of the design decisions taken during its development. Our implementation of CIFF has been inspired by work in the Automated Reasoning community on so-called *lean* theorem provers [1]. Our proof procedure manipulates a list of formulas rather than submitting these formulas *themselves* to the Prolog interpreter. One advantage of this approach is, for instance, that we can *report* variable substitutions at the meta-level rather than having Prolog making the actual instantiations (which would be problematic as CIFF computes only restrictions on the answer substitution, rather than an actual substitution).

The proof rules are repeatedly applied to the current node. Whenever a disjunction is encountered, it is split into a set of successor nodes (one for each disjunct). The procedure then picks one of these successor nodes to continue the proof search and backtracking over this choicepoint results in all possible successor nodes being explored. In theory, the choice of which successor node to explore next is taken nondeterministically; in practice we simply move through nodes from left to right. The procedure terminates when no more proof rules apply (to the current node) and finishes by extracting an answer from this node. Enforced backtracking will result in the next branch (if any) of the proof tree being explored, i.e. in any remaining abductive answers being enumerated. The Prolog predicate implementing the proof rules has the following form:

```
sat( +Node, +EV, +CL, +LM, +Defs, +FreeVars, -Answer).
```

`Node` is a list of goals, representing a conjunction. `EV` is used to keep track of existentially quantified variables in the node. This set is relevant to assess the applicability of some of the proof rules. `CL` (for constraint list) is used to store the

constraints that have been accumulated so far. The next argument, LM (for loop management), is a list of expressions of the form $A\!:\!B$ recording pairs of formulas that have already been used with particular proof rules, thereby allowing us to avoid loops that would result if these rules were applied over and over to the same arguments (this is necessary, for instance, for the *propagation* rule). This information can also be exploited to improve efficiency by identifying redundant proof steps. Defs is the list of iff-definitions in the theory. FreeVars is used to store the list of free variables. Finally, running sat/7 will result in the variable Answer to be instantiated with a representation of the abductive answer found by the procedure.

Each proof rule corresponds to a Prolog clause in the implementation of sat/7. For example, the *unfolding rule for atoms* is implemented as follows:

```
sat( Node, EV, CL, LM, Defs, FreeVars, Answer) :-
  member( A, Node), is_atom( A), get_def( A, Defs, Ds),
  delete( Node, A, Node1), NewNode = [Ds|Node1], !,
  sat( NewNode, EV, CL, LM, Defs, FreeVars, Answer).
```

The auxiliary predicate is_atom/1 will succeed whenever the argument represents an atomic goal. Furthermore, get_def(A,Defs,Ds), with the first two arguments being instantiated at the time of calling the predicate, will instantiate Ds with the list of lists representing the disjunction that defines the atom A according to the iff-definitions in Defs whenever there *is* such a definition (i.e. the predicate will fail for abducibles). Once get_def(A,Defs,Ds) succeeds we definitely know that the unfolding rule is applicable: there exists an atomic conjunct A in the current Node and it is not abducible. The cut in the penultimate line is required, because we do not want to allow any backtracking over the *order* in which rules are being applied. After we are certain that this rule should be applied we manipulate the current Node and generate its successor NewNode. We first delete the atom A and then replace it with the disjunction Ds. The predicate sat/7 then recursively calls itself with the new node.

3.4 Testing

The Prolog clauses in the implementation of sat/7 may be reordered almost arbitrarily (the only requirement is that the clause used to implement answer extraction is listed last). Each order of clauses corresponds to a different proof strategy, as it implicitly assigns different priorities to the different proof rules. This feature of our implementation allows for an experimental study of which strategies yield the fastest derivations. We hope to be able to exploit this feature of the implementation in our future work to study possible optimisation techniques. The order in which proof rules are applied in the current implementation follows some simple heuristics. For instance, logical simplification rules as well as rules to rewrite equality atoms are always applied first. Splitting, on the other hand, is one of the last rules to be applied.

The implementation of the CIFF proof procedure has been tested successfully on a number of examples. Most of these examples are taken from applications

$holds(F, T_2) \leftarrow happens(A, T_1),\ initiates(A, T_1, F),\ not\ clipped(T_1, F, T_2),\ T_1 < T_2$
$holds(F, T) \leftarrow initially(F),\ not\ clipped(0, F, T),\ 0 < T$
$holds(\neg F, T_2) \leftarrow happens(A, T_1),\ terminates(A, T_1, F),\ not\ declipped(T_1, F, T_2),\ T_1 < T_2$
$holds(\neg F, T) \leftarrow initially(\neg F),\ not\ declipped(0, F, T),\ 0 < T$

$clipped(T_1, F, T_2) \leftarrow happens(A, T),\ terminates(A, T, F),\ T_1 \leq T < T_2$
$declipped(T_1, F, T_2) \leftarrow happens(A, T),\ initiates(A, T, F),\ T_1 \leq T < T_2$
$happens(A, T) \leftarrow assume_happens(A, T)$

Table 2. Domain-independent rules in P_{plan}

$holds(F, T),\ holds(\neg F, T) \Rightarrow false$
$assume_happens(A, T),\ precondition(A, P) \Rightarrow holds(P, T)$
$assume_happens(A, T_2),\ not\ executed(A, T_2),\ time_now(T_1) \Rightarrow T_1 < T_2$

Table 3. Domain-independent integrity constraints in I_{plan} (for complete planning)

of CIFF within the SOCS project, which investigates the application of computational logic-based techniques to multiagent systems (e.g. the implementation of an agent's planning and a reactivity capabilities). While these are encouraging results, it should also be noted that this is only a first prototype and more research into proof strategies for CIFF as well as a fine-tuning of the implementation are required to achieve satisfactory runtimes for larger examples.

4 An Application to Abductive Planning

In this section, as an example application of the CIFF system, we consider how it can be used for planning. For this purpose we propose an abductive version of the event calculus. The event calculus is a formalism for reasoning about events (or actions) and change formulated by Kowalski and Sergot [18]. Since its publication a number of abductive variants of it have been proposed in the planning and abduction literature [23, 25, 26]. Our formulation is a novel variant, in part inspired by the $\mathcal{E}$-language [16], to allow situated, resource-bounded agents to generate partial plans in a dynamic environment, possibly inhabited by other agents. Partial planning is useful for two reasons. Firstly, it allows the agents to interleave planning, sensing and acting. Secondly, it prevents agents from spending time and effort constructing complete plans that may become unnecessary or unfeasible when they get round to executing them.

4.1 An Abductive Formulation of the Event Calculus

We model a planning problem within the framework of the event calculus (EC) in terms of a (non-allowed) abductive logic program with constraints $KB_{plan} = \langle P_{plan}, I_{plan}, A_{plan} \rangle$. In a nutshell, the EC allows us to write meta-logic programs which "talk" about object-level concepts of *fluents*, *actions*, and *time points*. The main meta-predicates of the formalism are: $holds(F, T)$ (fluent

F holds at time T), *clipped*(T_1, F, T_2) (fluent F is clipped —from holding to not holding— between times T_1 and T_2), *declipped*(T_1, F, T_2) (fluent F is declipped —from not holding to holding— between times T_1 and T_2), *initially*(F) (fluent F holds from the initial time, say time 0), *happens*(A, T) (action A occurs at time T), *initiates*(A, T, F) (fluent F starts to hold after action A at time T) and *terminates*(A, T, F) (fluent F ceases to hold after action A at time T). Roughly speaking, in a planning setting the last two predicates represent the cause-and-effects links between actions and fluents in the modelled world. We will also use a meta-predicate *precondition*(A, F) (the fluent F is one of the preconditions for the executability of action A). In our KB_{plan}, we allow fluents to be positive (F) or negative ($\neg F$). Our formulation of the EC also contains the predicates *observed*/2, *executed*/2, *time_now*/2, *assume_holds*/2, and *assume_happens*/2, which will be described shortly.

We now define KB_{plan}. To this end we first show the KB_{plan} that would be used for complete planning (but by situated agents, interacting with their environment) and then show how it can be modified to allow for partial planning. P_{plan} consists of three parts: *domain-independent rules*, *domain-dependent rules*, and a *narrative part*. I_{plan} consists of *domain-independent integrity constraints* and possibly *domain-dependent integrity constraints*. The basic rules and integrity constraints for the domain-independent part, adapted from the original EC, are shown in Tables 2 and 3. The only abducible predicate here is *assume_happens*. The reason why we do not use the *happens* predicate as an abducible and instead define it in terms of *assume_happens* will become clear when we describe the (domain-independent) bridging rules given in Table 4. The first of the integrity constraints given in Table 3 states that a fluent and its negation cannot hold at the same time, while the second one expresses that when assuming (planning) that some action will happen, we need to enforce that each of its preconditions hold. The third constraint ensures that the actions in the resulting plan that have not been executed yet are scheduled for the future.

The *domain-dependent rules* define the predicates *initiates*, *terminates*, and *precondition* (as well as any other predicates required by the modelling of the concrete domain). An example is given in Table 6 (see also Section 4.3). The first two rules state that catching a train or driving to a destination X initiates being at X. The two *initially*-facts state that initially A has petrol but does not have any anti-freeze. The last integrity constraint has the flavour of a *reactive rule* and it states that if you are planning to drive somewhere and you have observed shortly before that it is snowing then you must make sure that you have anti-freeze. The other rules and constraints are self-explanatory.

The *narrative part* of P_{plan} defines the predicates *initially*, *observed* and *executed*. For instance, *initially*$(at(bob, (1, 1)))$ expresses that agent *bob* is initially at location $(1, 1)$; *executed*$(go(bob, (1, 1), (2, 2), 3), 5)$ says that *bob* went from location $(1, 1)$ to location $(2, 2)$ at time 3, and this has been observed at time 5 by the agent who contains it in its P_{plan}; and *observed*$(at(jane, (2, 2)), 5)$ states that *jane* is observed to be at location $(2, 2)$ at time 5. The narrative part is not only domain-dependent, but it also refers to a particular "running scenario", in

$holds(F, T_2)$	$\leftarrow observed(F, T_1),\ not\ clipped(T_1, F, T_2),\ T_1 \leq T_2$
$holds(\neg F, T_2)$	$\leftarrow observed(\neg F, T_1),\ not\ declipped(T_1, F, T_2),\ T_1 \leq T_2$
$clipped(T_1, F, T_2)$	$\leftarrow observed(\neg F, T),\ T_1 \leq T < T_2$
$declipped(T_1, F, T_2)$	$\leftarrow observed(F, T),\ T_1 \leq T < T_2$
$happens(A, T)$	$\leftarrow observed(A, T)$
$happens(A, T)$	$\leftarrow executed(A, T)$

Table 4. Domain-independent bridging rules in P_{plan}

$holds(F, T),\ assume_holds(\neg F, T) \Rightarrow false$
$assume_holds(F, T),\ holds(\neg F, T) \Rightarrow false$
$assume_happens(A, T),\ precondition(A, P) \Rightarrow assume_holds(P, T)$

Table 5. Domain-independent integrity constraints in I_{plan} (for partial planning)

some concrete circumstances. Typically, *executed* facts within the narrative part of KB_{plan} of an agent refer to actions executed by the agent, whereas *observed* facts refer to properties of the environment, via facts of the form $observed(L, T)$ (the fluent literal L has been observed to hold at time T) as well as actions executed by other agents, via facts of the form $observed(A, T)$ (the action A has been observed to have happened at time T). The parameters of A can contain the identification of the agent who has executed the action. To accommodate observations, we add the *bridging rules* shown in Table 4 to P_{plan}. Note that the narrative part of P_{plan} changes over the life time of the agent (whereas the other parts of the knowledge base remain fixed).

Planning is done by reasoning with P_{plan} and I_{plan} to generate appropriate instances of the abducible predicate *assume_happens*, the successful execution of whose actions would establish the planning goal. Note that we need such a predicate (and *happens* cannot be used directly), because abducible predicates cannot be defined in the theory.

Now we show how KB_{plan} can be modified to facilitate partial planning. A partial plan contains subgoals as well as actions. Such subgoals are modelled using an additional abducible predicate *assume_holds*. Table 5 lists the additional integrity constraints in KB_{plan} pertaining, specifically, to partial planning. The purpose of the first two of the additional integrity constraints is to make sure that no fluent is assumed to hold at a time when the contrary of the fluent holds. The final constraint replaces the second constraint in Table 3.

This formulation allows the agent to plan for its goals by generating actions and subgoals that correspond to the preconditions of the actions, all of which are consistent with one another and with the observations that have been made. (An alternative formulation for partial planning can add the integrity constraints of Table 5 without modifying those in Table 3, but instead adding the following rule to P_{plan}: $holds(F, T) \leftarrow assume_holds(F, T)$. This would allow a more “liberal” way of partial planning, but for computational reasons we have chosen the first approach in the implementation.)

To summarise, our abductive formulation of the EC for partial planning consists of $KB_{plan} = \langle P_{plan}, I_{plan}, A_{plan} \rangle$, where

- P_{plan} consists of the rules in tables 2 and 4 and any *domain-dependent rules*,
- I_{plan} consists of the integrity constraints in Table 5 and the first and last constraints in Table 3 and any *domain-dependent integrity constraints*, and
- A_{plan} consists of the predicates *assume_happens* and *assume_holds*.

Intuitively, our proposal adds to more conventional abductive EC theories for planning the possibility to interact with the environment, by observing that properties hold and that other agents have executed actions. These additions pave the way to the situatedness of the agent in the environment, when planning is performed within a *"sense-plan-act"* cycle. Indeed, $observed(L, T)$ and $observed(A, T)$ facts (where A is an action executed by some other agent) are added to the narrative part of KB_{plan} as the result of a sensing operation of the agent, whereas $executed(A, T)$ facts (where A is an action executed by the agent) are added as the result of the execution of a planned action in the environment. Each step in the cycle takes place at some concrete time, used to time-stamp the facts recorded in the narrative part of KB_{plan}.

4.2 Goals and Partial Plans in CIFF

Here we describe in more detail how goals and partial plans of an agent are specified within the framework of the EC. A *goal* is a conjunction of the form $holds(L, T) \wedge TC$, where TC is a set of constraints on T, referred to as the *temporal constraints* of the goal. A *(partial) plan* for a given goal consists of

- a (possibly empty) set of atoms of the form $assume_happens(A, T) \wedge TC$, where TC is a set of constraints on T, referred to as the *Actions* in the plan;
- a (possibly empty) set of atoms of the form $assume_holds(L, T) \wedge TC$, where TC is a set of constraints on T, referred to as the *SubGoals* in the plan.

A partial plan for a set of goals is a set of partial plans, one for each individual goal. Goals, actions, subgoals and temporal constraints will be typically non-ground, and the variables occurring in them are implicitly existentially quantified over the set of (partial) plans and goals.

Given a set of goals, GS, where each goal is of the form $holds(L, T) \wedge TC$, a (possibly empty) set of subgoals *SubGoals*, a (possibly empty) set of actions *Actions* (representing already existing partial plans that we are trying to expand), and a (possibly empty) set of temporal constraints, to compute a partial plan for GS at a time τ, CIFF uses the program $\langle Th, I \rangle$, where Th is a set of iff-definitions formed by the selective completion of P_{plan}, given A_{plan}, and I is the set of all integrity constraints in I_{plan}.

To choose a query for CIFF, we first have to select the goals to be planned for in the next round of planning. Let *gs(Goals, Time)* be a goal *selection function* that takes as input a set of goals *Goals* of the form $holds(L, T) \wedge TC$ and a time of evaluation *Time*, and returns as output a subset of *Goals*. We do not give a definition for such a selection function here. A number of different definitions

$$initiates(catch_train_to(X), T, at(X))$$
$$initiates(drive_to(X), T, at(X))$$
$$initiates(fill_up(X), T, have(X)) \leftarrow X = petrol$$
$$initiates(fill_up(X), T, have(X)) \leftarrow X = anti_freeze$$
$$initially(\neg have(anti_freeze))$$
$$initially(have(petrol))$$
$$precondition(drive_to(X), have(petrol))$$
$$assume_happens(catch_train_to(X), T),\ holds(train_strike, T) \Rightarrow false$$
$$assume_happens(drive_to(X), T_1),\ observed(snowing, T_2),\ T_1 - 5 \leq T_2 \leq T_1 \Rightarrow assume_holds(have(anti_freeze, T_1))$$

Table 6. Domain-dependent rules in KB_{plan} for the airport example

are possible, for example gs can select those goals that are deemed "urgent" according to some measure of urgency (e.g. how close their times are to $Time$). Let $GS' = GS \cup \{holds(L, T) \wedge TC \mid assume_holds(L, T) \wedge TC \in SubGoals\}$. Let $gs(GS', \tau) = SelectedGoals$. CIFF is then invoked with a query including:

(1) all selected goals: the conjunction of all atoms $holds(L, T) \wedge TC$, for all the goals in $SelectedGoals$;
(2) all non-selected (sub)goals: the conjunction of all $assume_holds(L', T') \wedge TC$ such that $holds(L', T') \wedge TC$ is in GS or $assume_holds(L', T') \wedge TC$ is in $SubGoals$, and $holds(L', T') \wedge TC$ is not in $SelectedGoals$;
(3) $time_now(\tau)$;
(4) the conjunction of all atoms $assume_happens(A', T') \wedge TC$ in $Actions$.

4.3 Example

To illustrate the application of CIFF to planning consider the following scenario. Agent A has the goal of being at the airport before time 10. It knows of two ways of getting there, by train and by car. It has already learned that there is a train strike. So it plans to drive there. Then it makes two observations, one that it is low on petrol, and, the other, that it is snowing. So it plans to get petrol to accommodate the first observation, and knowing that it is short on anti-freeze it reacts to the information about the snow by adding a goal of topping up its anti-freeze (thus repairing its plan to fit in with its changed environment). So through the cycle of observations and (partial) planning it finally constructs a complete plan consisting of the three actions of filling up petrol, topping up anti-freeze, and driving to the airport at appropriate times. The domain-dependent and narrative parts of agent A's KB_{plan} are shown in Table 6.

Suppose we are now at time 4 and we have the goal of being at the airport before time 10. The goal given to CIFF would be the following:

$$holds(at(airport), T),\ T < 10,\ time_now(4)$$

CIFF will return the following partial plan (with $4 < T' < 10$):

$$assume_holds(have(petrol), T'),\ assume_happens(drive_to(airport), T')$$

At this stage, now at time 5, suppose we make the two observations $observed(\neg have(petrol), 5)$ and $observed(snowing, 5)$. These are recorded in agent A's P_{plan}. For the next round of planning, at time 6, say, CIFF can be called with the following set of goals:

$$holds(have(petrol), T'),\ assume_happens(drive_to(airport), T'),\\ 4 < T',\ T' < 10,\ time_now(6)$$

CIFF will then augment the existing partial plan by adding to it the following new subgoals and actions (with the additional constraints $6 < T''$ and $T'' < T'$):

$$assume_holds(have(anti_freeze), T'),\ assume_happens(fill_up(petrol), T'')$$

At time 7, say, if there are no further observations, CIFF will complete the plan by adding the action $fill_up(anti_freeze)$ with appropriate time constraints.

4.4 Implementation of the Planner

Given the computational model for planning put forward above and our implementation of the CIFF proof procedure described in Section 3, the implementation of a simple abductive planner has been straightforward. In essence, it consists of only a single Prolog clause:

```
plan( Narration, Assumptions, Goals, TCs, Answer) :-
  kbplan( PlanDefs, ICs),
  close_pred( executed/2, Narration, ExecActions),
  close_pred( observed/2, Narration, Observations),
  close_pred( time_now/1, Narration, TimeNow),
  Defs = [ExecActions,Observations,TimeNow|PlanDefs],
  append( Goals, TCs, Goals1), append( Goals1, Assumptions, Query),
  ciff( Defs, ICs, Query, Plan:Substitution:NewTCs),
  delete_list( Plan, Assumptions, NewPlan),
  Answer = NewPlan:Substitution:NewTCs.
```

`Narration` is a list of (ground) terms of the form `executed(Action,T)`, `observed(Fluent,T)`, and `observed(Action,T)`, representing the narrative part of KB_{plan}, as well as a single term of the form `time_now(N)` to communicate the current time (`N` is an integer). `Assumptions` is a list of terms of the form `assume_holds(Goal,T)` and `assume_happens(Action,T)` encoding the goals and actions in the current partial plan. `Goals`, the goals to plan for, is a list of terms of the form `holds(Goal,T)` and `TCs` is a list of temporal constraints over variables occurring in the goals and actions given in the input. The variable `Answer` will be instantiated with a representation of the chosen plan if there exists one; otherwise the call to `plan/5` will fail.

In the first subgoal of the implementation of `plan/5`, the `kbplan/2` predicate is used to retrieve the iff-definitions (`PlanDefs`) and the integrity constraints (`ICs`) in the non-narrative parts of KB_{plan} (we assume these have been asserted earlier). The predicate `close_pred/3` is used to generated iff-definitions for the

predicates occurring in the `Narration` and these are appended to the list of definitions to obtain `Defs`. Then the CIFF proof procedure is called with `Defs` as the background theory, `ICs` as the integrity constraints, and the list of all other relevant terms as the query. The first component of the answer consists of a list of abducible predicates encoding the plan (using `assume_holds/2` for subgoals and `assume_happens/2` for actions). To simplify the output, the assumptions (goals and actions) already present in the input are then deleted from this list. Furthermore, the answer may also include a list of restrictions on the answer substitution (`Substitution`) and an updated list of constraints (`NewTCs`).

5 Conclusion

We have presented a Prolog implementation of the CIFF procedure that extends the general purpose abductive proof procedure IFF by dealing with non-allowed programs and by handling constraints. The implementation allows the use of any abductive logic program and presents answers in the form of abducibles with instantiations and restrictions of variables. We have also discussed an application to planning where, by varying the input theory, we can construct complete or partial plans in the presence or absence of narrative information.

The CIFF system has been used extensively in the PROSOCS framework [27] to implement a planning component as well as for reactivity and temporal reasoning. Reactivity allows condition-action rule behaviour used in PROSOCS, for example, for plan repair, for strategies for negotiation and communication with other agents, and generally for reacting to changes in the environment. The temporal reasoning capability is based on an extensive abductive logic program based on the event calculus that deals with the revision of the agent's knowledge as a result of assimilating new information from its environment, including other agents [2]. We have been able to use and test the CIFF system on a number of examples for all three applications without any modifications. While so far, we have concentrated on providing an implementation of CIFF that correctly implements the semantics, that is easy to understand, and that supports future extensions, in our future work we hope to also study possible optimisation techniques for CIFF.

Acknowledgements. This work was supported by the European Commission FET Global Computing Initiative, within the SOCS project (IST-2001-32530). We thank all SOCS partners for useful feedback and in particular Marco Gavanelli and Michela Milano for suggestions on implementing the constraint solver.

References

1. B. Beckert and J. Posegga. leanTAP: Lean, Tableau-based deduction. *J. Automated Reasoning*, 15(3):339–358, 1995.
2. A. Bracciali and A. C. Kakas. Frame consistency: Computing with causal explanations. In *Proc. NMR-2004*, 2004. To appear.

3. M. Carlsson, G. Ottosson, and B. Carlson. An open-ended finite domain constraint solver. In *Proc. PLILP-97*, 1997.
4. K. L. Clark. Negation as failure. In *Logic and Data Bases*. Plenum Press, 1978.
5. L. Console, D. T. Dupré, and P. Torasso. On the relationship between abduction and deduction. *J. Log. Computat.*, 1(5):661–690, 1991.
6. M. Denecker and D. De Schreye. SLDNFA: An abductive procedure for abductive logic programs. *J. Log. Prog.*, 34(1):111–167, 1998.
7. M. Denecker and B. Van Nuffelen. Experiments for integration CLP and abduction. In *Proc. Workshop on Constraints*, 1999.
8. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The CIFF proof procedure: Definition and soundness results. Technical Report 2004/2, Department of Computing, Imperial College London, 2004.
9. K. Eshghi and R. A. Kowalski. Abduction compared with negation by failure. In G. Levi and M. Martelli, editors, *Proc. ICLP-1989*. MIT Press, 1989.
10. T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *J. Log. Prog.*, 33(2):151–165, 1997.
11. J. Jaffar and M. J. Maher. Constraint logic programming: A survey. *J. Log. Prog.*, 19-20:503–582, 1994.
12. A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.
13. A. C. Kakas and P. Mancarella. Abductive logic programming. In *Proc. Workshop Logic Programming and Non-Monotonic Logic*, 1990.
14. A. C. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In *Proc. ECAI-2004*, 2004. To appear.
15. A. C. Kakas, A. Michael, and C. Mourlas. ACLP: Abductive constraint logic programming. *J. Log. Prog.*, 44:129–177, 2000.
16. A. C. Kakas and R. Miller. A simple declarative language for describing narratives with actions. *J. Log. Prog.*, 31(1–3):157–200, 1997.
17. A. C. Kakas, B. Van Nuffelen, and M. Denecker. A-system: Problem solving through abduction. In *Proc. IJCAI-2001*. Morgan Kaufmann, 2001.
18. R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.
19. R. A. Kowalski, F. Toni, and G. Wetzel. Executing suspended logic programs. *Fundamenta Informaticae*, 34:203–224, 1998.
20. K. Kunen. Negation in logic programming. *J. Log. Prog.*, 4:289–308, 1987.
21. F. Lin and J.-H. You. Abduction in logic programming: A new definition and an abductive procedure based on rewriting. *Artif. Intell.*, 140(1/2):175–205, 2002.
22. P. Mancarella and G. Terreni. An abductive proof procedure handling active rules. In *Proc. AI*IA-2003*. Springer-Verlag, 2003.
23. L. Missiaen, M. Bruynooghe, and M. Denecker. CHICA, an abductive planning system based on event calculus. *J. Log. Computat.*, 5(5):579–602, 1995.
24. F. Sadri and F. Toni. Abduction with negation as failure for active and reactive rules. In *Proc. AI*IA-1999*. Springer-Verlag, 2000.
25. M. P. Shanahan. Prediction is deduction but explanation is abduction. In *Proc. IJCAI-1989*. Morgan Kaufmann, 1989.
26. M. P. Shanahan. *Solving the Frame Problem*. MIT Press, 1997.
27. K. Stathis, A. C. Kakas, W. Lu, N. Demetriou, U. Endriss, and A. Bracciali. PROSOCS: A platform for programming software agents in computational logic. In *Proc. AT2AI-2004*, 2004.
28. Swedish Institute of Computer Science. *Sicstus Prolog User Manual*, 2003.

A multi context-based Approximate Reasoning

Luciano Blandi and Maria I. Sessa

Dipartimento Matematica e Informatica
Universita' di Salerno
via Ponte Don Melillo - 84084 Fisciano (SA), Italy
{lblandi, misessa}@unisa.it

Abstract. The paper concerns with the theory of similarity relations in the framework of Logic Programming. Similarity relation [34] is a formal notion that allows us to manage alternative instances of entities that can be considered 'equal' with a given degree. In [8, 15, 30, 29, 18] this notion has been encapsulated in an inference engine, based on the Logic Programming paradigm, modifying the inference model to provide a more flexible unification (approximately equal) than the dichotomic match (equal or not). In many situations, more than one similarity relation can be defined in a universe by taking into account different contexts. In this paper we studied some aggregations of such fuzzy relations and their application to our extented Logic Programming framework.

1 Introduction and previous works

The main interest in Logic Programming field traditionally concerns problems related to the analysis and the efficiency of exact inferences allowed by logic programs [1, 14]. However, very often the need of methods to enhance this capability, in order to deal with approximate information or flexible inference schemes, arises in many applications. In general, approximate reasoning capabilities are introduced in the Logic Programming framework by considering the inference system based on fuzzy logic rather than on conventional two-valued logic.

In [8] a methodology that allows to manage uncertain and imprecise information in the frame of the declarative paradigm of Logic Programming has been proposed. With this aim, a Similarity relation R between function and predicate symbols in the language of a logic program is considered. Approximate inferences are then possible since Similarity relation allows us to manage alternative instances of entities that can be considered "equal" with a given degree.

With respect to the previous literature [4, 31, 12, 13], this approach is very different since the approximation is represented and managed at a syntactic-level, instead of at a rule-level. Roughly speaking, the basic idea is that the fuzziness feature is provided by an abstraction process which exploits a formal representation of similarity relations between elements in the alphabet of the language (constants, functions, predicates). On the contrary, in the underground logic theory, the inference rule as well as the usual crisp representation of the considered universe are not modified. It allows us to avoid both the introduction of weights on the clauses, and the use of fuzzy sets as elements of the language.

In [29] the operational counterpart of this extension is faced by introducing a modified SLD Resolution procedure. Such a procedure allows us to compute numeric values belonging to the interval [0,1] providing an approximation measure of the obtained solutions. These numeric values are computed through a generalized unification mechanism. In [18] a Prolog interpreter written in Java which implements this Similarity-based extension has been presented.

In these works, the approximation was based on a single similarity relation. However, by taking into account different contexts (i.e. points of view), it is possible to define different similarity relations in a given universe. Let us consider the following Example

Example 1.

Let U be a set of animals denoted with

$U = \{M,\ B,\ G,\ E,\ P,\ H,\ S,\ T,\ C,\ W,\ D\}$

where these letters stand for

$M = man,\ \ B = bear,\ \ G = gorilla,\ \ E = eagle,\ P = pigeon,\ \ H = hawk,$

$S = shark,\ \ T = tiger,\ C = cat,\ \ W = wolf,\ \ D = dog.$

We can define a similarity R_1 between elements in U based on feature 'morphology' by setting for any $x, y \in U$

$R_1(x, y) = R_1(y, x)$

$R_1(x, y) = 1 \quad if\ \ x = y$

$R_1(G, M) = R_1(D, W) = R_1(E, H) = .8$

$R_1(W, T) = R_1(D, T) = .6$

$R_1(C, T) = R_1(C, W) = R_1(C, D) = .4$

$R_1(B, T) = R_1(B, C) = R_1(B, W) = R_1(B, D) = R_1(E, P) = R_1(P, H) = .2$

$R_1(x, y) = 0 \quad otherwise.$

Also, we can define a similarity R_2 between elements in U based on feature 'aggressiveness' by setting for any $x, y \in U$

$R_2(x, y) = R_2(y, x)$

$R_2(x, y) = 1 \quad if\ \ x = y$

$R_2(G, W) = R_2(T, S) = R_2(E, H) = .8$

$R_2(M, D) = R_2(B, S) = R_2(B, T) = R_2(C, D) = .6$

$R_2(M, E) = R_2(M, H) = R_2(M, C) = R_2(B, G) = R_2(B, W) = R_2(G, S) =$
$= R_2(G, T) = R_2(E, C) = R_2(E, D) = R_2(H, C) = R_2(H, D) = R_2(S, W) = R_2(T, W) = .4$

$R_2(M, B) = R_2(M, G) = R_2(M, S) = R_2(M, T) = R_2(W, M) = R_2(B, E) =$
$= R_2(B, H) = R_2(B, C) = R_2(B, D) = R_2(G, E) = R_2(G, H) = R_2(G, C) = R_2(G, D) =$
$= R_2(E, S) = R_2(E, T) = R_2(E, W) = R_2(H, S) = R_2(H, T) = R_2(H, W) = R_2(S, C) =$
$= R_2(S, D) = R_2(T, C) = R_2(T, D) = R_2(C, W) = R_2(W, D) = .2$

$R_2(x, y) = 0 \quad otherwise.$

In many cases, we may need to aggregate different relations. Aggregation of binary (fuzzy) relations is an important and challenging mathematical problem in applied areas as social choice, group choice, multiple-criteria decision-making, synthesis of implication functions, etc. Formally, this problem can be formulated in terms of group choice theory as follows: suppose U is a finite set of alternatives and $\mathcal{R} =< R_1, ..., R_n >$ is an ordered n-tuple of binary (fuzzy) relations on U. Elements of $\mathcal{R}$ are regarded as individual preferences and $\mathcal{R}$ is called a *profile* of individual preferences on the set U of alternatives. For a given U, an aggregation rule assigns a group preference R to each profile $\mathcal{R}$ of individual preferences on U (very often it is assumed that $n > 2$). We denote this rule by the same letter R. Depending on the application area, various restrictions are imposed on R. In [5, 11, 2, 26, 27, 21, 28], the fuzzy binary relations $R_1, ..., R_n$ and R satisfy T-transitivity property in which T is an Archimedian t-norm. In our framework we consider transitivity based on the minimum triangular norm. In particular, in this paper we study properties of $R_{min} = \bigcap_i R_i$ and $R_{max} = \bigcup_i R_i$ assuming that individual and group preferences are similarity relations on a finite set of alternative U.

The paper is organized as follows. After preliminaries on similarity relation in Logic Programming framework, Section 3 will study the aggregation of similarity relations by intersection (Subsection 3.1) and union (Subsection 3.2), and their exploitation in the similarity based Logic Programming (Subsection 3.3). The last section contains some concluding remarks.

2 Preliminaries

2.1 Similarity relation

An important and very intuitive theoretical basis for fuzzy subsets is given by the concept of fuzzy equivalence relations, which, in some sense, measure the degree to which two points of the universe are indistinguishable, and which generalize and relax the concept of classical equivalence relations. A strong motivation for this notion follows from the so-called Poincaré Paradox [25]: if for three (real world) objects A, B and C we know that A is indistinguishable from B and B is indistinguishable from C, we cannot necessarily conclude that A is indistinguishable from C too. Fuzzy equivalence relations have been introduced under the name of similarity relation in [34] (with respect to the minimum T_M, the generalization to t-norms was considered in [32]). In this section, we will recall some well-known definitions and properties related to similarity relation and to its application in the Logic Programming framework.

In Cantorian set theory, a relation on a universe U can be identified with a subset of U^2. By analogy, a fuzzy relation on U is then a fuzzy subset of U^2. For early traces of properties of fuzzy relations see [34] and [22–24], more recent treatments include [6, 3].

At first, let us recall that a *T-norm* is a binary operation $\wedge : [0,1] \times [0,1] \to [0,1]$ associative, commutative, non-decreasing in both the variables, and such that $x \wedge 1 = 1 \wedge x = x$ for any x in [0,1]. In the sequel, we assume that $x \wedge y$ is the *minimum* between the two elements $x, y \in [0,1]$.

Definition 1. *Given a T-norm, a fuzzy relation R on a set U is* T-transitive *if and only if $T(R(x,y), R(y,z)) \leq R(x,z)$ for any $x, y, z \in U$.*

Among all T-transitive fuzzy relations, similarity relations and fuzzy T-preorders are the most important ones.

Definition 2. *A* similarity *on a domain U is a fuzzy relation $R : U \times U \to [0,1]$ in U such that the following properties hold*

i) $R(x,x) = 1$ *for any* $x \in U$ *(reflexivity)*

ii) $R(x,y) = R(y,x)$ *for any* $x, y \in U$ *(symmetry)*

iii) $R(x,z) \geq R(x,y) \wedge R(y,z)$ *for any* $x, y, z \in U$ *(transitivity)*

we say that R is strict *if the following implication is also verified*

iv) $R(x,z) = 1 \implies x = z$.

The value $R(x,z)$ can be interpreted as the degree of equality or the degree of indistinguishability of x and y or, equivalently, as the truth value of the statement 'x is equal to y'. The $\wedge$-transitivity is a many-valued model of the proposition 'IF x is equal to y AND y is equal to z THEN x is equal to z'.

Similarities also are called indistinguishability operators, fuzzy equalities, fuzzy equivalences, likeness, probabilistic relations, proximity relations, M-valued equality, depending on the authors and on the t-norm used to model their transitivity.

There is a lot of work around this concept and it has been proved to be a useful tool both in the theoretical aspects of fuzzy logic and in their applications such as fuzzy control or approximate reasoning.

2.2 Similarity relation and closure operators

We synthetically give some well known notions concerning closure operators and equivalence relations.

Definition 3. *Let $(P, \preceq)$ be a poset. An operator $H : P \to P$ is called a* closure *(resp.* reductive*) operator if for any x, y in P the following properties hold:*

i) $x \preceq H(x)$ $\qquad$ *(resp.* $H(x) \preceq x$*)*

ii) $H(H(x)) = H(x)$

iii) $x \preceq y \Longrightarrow H(x) \preceq H(y)$.

Proposition 1. *Let $\equiv$ be an equivalence relation on a set S and $\mathcal{P}(S)$ the powerset of S. Then, the operator $H_{\equiv} : \mathcal{P}(S) \to \mathcal{P}(S)$ such that for any $X \subseteq \mathcal{P}(S)$*

$$H_{\equiv}(X) = \{x' \in S \quad | \quad \exists x \in X : \quad x' \equiv x)\}$$

is a closure operator.

The following notion of $\lambda-cut$ is crucial in fuzzy set theory:

Definition 4. *Let U be a domain and $R : U \times U \to [0, 1]$ a fuzzy relation in U. Then, for any $\lambda \in [0, 1]$, the relation $\approxeq_{R,\lambda}$ in U defined as*

$$x \approxeq_{R,\lambda} y \qquad \Longleftrightarrow \qquad R(x, y) \succeq \lambda$$

is named cut of level λ *(in short* λ-cut*) of R.*

Similarity relations are strictly related with equivalence relations and, then, to closure operators. Indeed, the notion of $\lambda - cut$ allows us to define a similarity by means of a suitable family of equivalence relations according to the following result that can be easily proven.

Proposition 2. *Let U be a domain and $R : U \times U \to [0, 1]$ a Similarity in U. Then, for any $\lambda \in [0, 1]$, the relation $\approxeq_{R,\lambda}$ in U is an equivalence relation. Also, the operator $H_{\approxeq_{R,\lambda}} : \mathcal{P}(U) \to \mathcal{P}(U)$ such that for any $X \in \mathcal{P}(U)$*

$$H_{\approxeq_{R,\lambda}}(X) = \{z \in U \mid \exists x \in X : \ x \approxeq_{R,\lambda} y\} = \{z \in U \mid \exists x \in X : \ R(z, x) \geq \lambda\},$$

is a closure operator.

Proposition 3. *Let R be a similarity in a domain U and, for any $\lambda \in [0, 1]$ let $\approxeq_{R,\lambda}$ be the $\lambda-cut$ of R. Then, $\{\approxeq_{R,\lambda}\}_{\lambda \in [0,1]}$ is a family of equivalence relations such that,*

i) for any μ and λ in $[0, 1]$, $\quad \lambda \preceq \mu \quad \Rightarrow \quad \approxeq_{R,\lambda} \supseteq \approxeq_{R,\mu}$

ii) for any μ in $[0, 1]$, $\quad \bigcap_{\lambda \preceq \mu} \approxeq_{R,\lambda} = \approxeq_{R,\mu}$.

Conversely, let $\{\approxeq_{\lambda}\}_{\lambda \in [0,1]}$ be a family of equivalence relations satisfying conditions i) and ii). Then the relation R defined by setting

$$R(x, y) = Sup\{\lambda \in [0, 1] \quad | \quad x \approxeq_{\lambda} y\}$$

is a similarity whose family of $\lambda-cuts$ is equal to the family $\{\approxeq_{\lambda}\}_{\lambda \in [0,1]}$.

Any $\lambda-$cut can be considered as a generalization of the equality. This notion plays an important rule in our approach. Indeed, the relation $\approxeq_{R,\lambda}$ formalizes the idea that two constant symbols can be considered equal with a fixed approximation level $\lambda \in [0, 1]$. Such a level provides a measure of the allowed approximation in order to avoid failure of matching between constant symbols in a SLD-derivation process.

2.3 Logic Programming with Similarity

We briefly recall that a logic program P is a set of universally quantified Horn clauses on a first order language L, denoted with $H \leftarrow B_1, \ldots, B_k$, and a goal is a negative clause, denoted with $A_1, \ldots, A_n$. We denote with B_L the set of ground atomic formulae in L, i.e. the Herbrand base of L, and with T_P the immediate consequence operator $T_P : \mathcal{P}(B_L) \mapsto \mathcal{P}(B_L)$ defined by:

$$T_P(X) = \{a | a \leftarrow a_1, \ldots, a_n \in \Gamma(P) \text{ and } a_i \in X, 1 \leq i \leq n\}$$

where $\Gamma(P)$ denotes the set of all ground instances of clauses in P. The application of Tarski's fixpoint theorem yields a characterization of the semantics of P, which is the least Herbrand model M_P of P given by:

$$M_P = lfp(T_P) = \bigcup_{n \geq 0} T_P^n(\emptyset)$$

where lfp stands for least fixpoint [1].

In the classical Logic Programming, function and predicate symbols of the language L are crisp elements, i.e., distinct elements represent distinct information and no matching is possible. In [8] the exact matching between different entities is relaxed by introducing a Similarity relation R in the set of constant, function and predicate symbols in the language of a logic program P. In order to deal with the approximation introduced by a similarity relation R, the program P is extended by adding new clauses which are "similar" at least with a fixed degree λ in (0,1] to the given ones. This program transformation is obtained by considering the closure operator H_λ associated to R. The new logic program

$$H_\lambda(\Gamma(P)) = \{C' \in L | \exists C \in \Gamma(P) \text{ such that } R(C, C') \geq \lambda\},$$

named *extended-program of level* λ, allows us to enhance the inference process.

An alternative way to manage the information carried on by the Similarity introduced between function and predicate symbols in P, is given by considering as an unique element different symbols which have Similarity degree greater or equal to λ. In other words, we consider the quotient set of $\approxeq_{R,\lambda}$ as a new alphabet L_λ, where $F/\approxeq_{R,\lambda}$ and $R/\approxeq_{R,\lambda}$ are the sets of function and predicate symbols, respectively. More formally, let us denote with $[s] \in L_\lambda$ the equivalence class of a symbol $s \in F \cup R$ with respect to $\approxeq_{R,\lambda}$We call *translation up* to $\approxeq_{R,\lambda}$ the function:

$$\tau_\lambda : F \cup R \mapsto F/\approxeq_{R,\lambda} \cup R/\approxeq_{R,\lambda}$$

defined by setting:

$$\tau_\lambda(x) = x \text{ for any variable } x \in V, \text{ and } \tau_\lambda(f) = [f]$$

for any function/predicate symbol $f \in F \cup R$. Recursively, we can easily define the extension of τ_λ to the sets of formulae in L. Let us consider a logic program P on the language L.

The set

$$P_\lambda = \tau_\lambda(\Gamma(P)) = \{C' \in L_\lambda | C' = \tau_\lambda(C),\ \ C\ \ clause\ \ in\ \ \Gamma(P)\}$$

is a logic program that we name *abstract-program* of level λ.

By considering the abstract program P_λ, it is possible to express information provided by the similarity relation in a syntectic way exploiting the quotient language L_λ. Then, P_λ could be used to manage similarity-based reasoning as well as $H_\lambda(\Gamma(P))$. In [30] the equivalence of these two approaches has been shown for first order languages by using an abstract interpretation technique.

It is worth to stress that the introduced similarity generally changes the semantic of the original program. Indeed, it allows us to add new clauses to P providing the extended program $H_\lambda(P)$. This is the more straight way to implement the approximated inference process based on similarity.

On the other hand, by considering the abstract program P_λ, it is possible to express information provided by the similarity relation in a syntectic way exploiting the quotient language L_λ.

Both these programs allow to perform approximate inferences by assuming a "tolerance" level $\lambda \in (0, 1]$ in the relaxed matching between different function/predicate symbols.

In [8], the formal notion of *fuzzy least Herbrand model* $M_{P,R} : B_L \mapsto [0, 1]$ of the program P with respect to the Similarity R is defined by setting for any $A \in B_L$:

$$\begin{aligned} M_{P,R}(A) &= Sup\{\lambda \in [0,1] \,|A \in M_{H_\lambda(\Gamma(P))}\} \\ &= Sup\{\lambda \in [0,1] \,|H_\lambda(\Gamma(P)) \vDash A\ \} \end{aligned}$$

Roughly speaking, for any $A \in B_L$ the value $M_{P,R}(A)$ provides the best deduction degree of A, i.e, the best level of approximation λ that allows us to prove A by considering an extended program $H_\lambda(\Gamma(P))$. It can be proved that:

$$\begin{aligned} M_{P,R}(A) &= Sup\{\lambda \in [0,1] \,|t_\lambda(A) \in M_{P_\lambda}\} \\ &= Sup\{\lambda \in [0,1] \,|P_\lambda \vDash \tau_\lambda(A)\} \end{aligned}$$

Thus, in order to compute the fuzzy least Herbrand model of a program P extended with a Similarity R, we can equivalently perform our computations in the extended or in the abstract domain.

3 On aggregations in multi context-based Logic Programming framework

In many situations, there can be more than one similarity relation defined in a universe. For example that we have a set of elements defined by some features. We can generate a similarity relation from each feature. In these cases, we must manage and use such information in an appropriated way, for instance we may need to aggregate the obtained relations. In this Section we give a first formal environment in order to make that.

3.1 Aggregation of similarity relations by intersection

The most common way to put together a family of T-transitive fuzzy relations is by calculating their minimum (or infimum), which also is a T-transitive relation. Indeed the following well-known proposition [33] states that x, y are related with respect to R if and only if they are related with respect to all the relations of the family (because the infimum is used to model the universal quantifier $\forall$ in fuzzy logic [9]).

Proposition 4. *Let $(R_i)_{i \in I}$ be a family of T-transitive fuzzy relations on a set U. The relation defined for all $x, y \in U$ by*

$$R(x, y) = min_{i \in I} R_i(x, y)$$

is a T-transitive fuzzy relation on U.

In particular,

Corollary 1. *Let $R_1, \ldots, R_n$ be n similarity relations on a set U. The relation $R_{min} = \bigcap_i R_i$ defined for all $x, y \in U$ by*

$$R_{min}(x, y) = min\{R_1(x, y), \ldots, R_n(x, y)\}$$

is a similarity relation on U.

Example 2. We consider the similarity relations R_1 and R_2 defined in Example 1. Then,

$R_{min}(x,y) = R_{min}(y,x)$

$R_{min}(x,y) = 1 \quad if \ x = y$

$R_{min}(E,H) = .8$

$R_{min}(W,T) = R_{min}(C,D) = .4$

$$R_{min}(G,M) = R_{min}(D,W) = R_{min}(D,T) = R_{min}(C,T) = R_{min}(C,W) = R_{min}(B,T) = \\ = R_{min}(B,C) = R_{min}(B,W) = R_{min}(B,D) = .2$$

$R_{min}(x,y) = 0 \quad otherwise.$

3.2 Aggregation of similarity relations by union

The binary fuzzy relation $R_{min} = \bigcap_i R_i$ is an extreme case of aggregation rule because it is very restrictive. Indeed, $R_{min}(a,b) \leq R_i(a,b)$ for any i. Many times this way to aggregate fuzzy relations by intersection leads to undesirable results in applications. The reason is that the minimum has a drastic effect. For instance, if two objects of our universe are very similar or indistinguishable for all but one similarity relation, and for this particular one the similarity value is very low, then the result applying the minimum will give this last measure and will lose the information of all the other ones. This can be reasonable and useful if we need a perfect matching with respect to all our relations, but this is not the case in many situations. When we need to take all the relations into account in a less drastic way, we need to use other ways to aggregate them. A possibility of softening the previous proposition is by replacing the intersection by the union.

We define $R_{max} = \bigcup_i R_i$ by setting $R_{max}(x,y) = max\{R_1(x,y), \ldots, R_n(x,y)\}$.

Note that R_{max} not is a similarity relation.

Example 3. We consider the similarity relations R_1 and R_2 defined in Example 1. Then,

$R_{max}(x,y) = R_{max}(y,x)$

$R_{max}(x,y) = 1 \quad if \ x = y$

$R_{max}(M,G) = R_{max}(G,W) = R_{max}(T,S) = R_{max}(W,D) = R_{max}(E,H) = .8$

$R_{max}(M,D) = R_{max}(B,S) = R_{max}(B,T) = R_{max}(C,D) = R_{max}(T,D) = .6$

$$R_{max}(C,W) = R_{max}(T,C) = R_{max}(M,E) = R_{max}(M,H) = R_{max}(M,C) = R_{max}(B,G) = \\ = R_{max}(B,W) = R_{max}(G,S) = R_{max}(G,T) = R_{max}(E,C) = R_{max}(E,D) = \\ = R_{max}(H,C) = R_{max}(H,D) = R_{max}(S,W) = R_{max}(T,W) = .4$$

$$R_{max}(M,B) = R_{max}(M,S) = R_{max}(M,T) = R_{max}(W,M) = R_{max}(B,E) = R_{max}(B,H) = \\ = R_{max}(P,E) = R_{max}(B,C) = R_{max}(B,D) = R_{max}(G,E) = R_{max}(G,H) = R_{max}(G,C) = \\ = R_{max}(P,H) = R_{max}(G,D) = R_{max}(E,S) = R_{max}(E,T) = R_{max}(E,W) = R_{max}(H,S) = \\ = R_{max}(H,T) = R_{max}(H,W) = R_{max}(S,C) = R_{max}(S,D) = .2$$

$R_{max}(x,y) = 0 \quad otherwise.$

Note that

$\mathrm{R_{max}(W, D) = 0.8, R_{max}(B, W) = 0.4, R_{max}(B, D) = 0.2}$

which violates the min-transitivity property because

$\mathrm{R_{max}(B, D) \ngeq R_{max}(W, D) \wedge R_{max}(B, W)}$

Let us recall that the max-T product [34, 33] allows us to construct the transitive closure of a given reflexive and symmetric fuzzy relation.

Definition 5. *Let T be a T-norm and R, S two fuzzy relations in a set U. The max-T (or sup-T) product $R \circ S$ of R and S is the fuzzy relation on U defined by*

$$(R \circ S)(x,y) = \sup_{z \in U} T(R(x,z), S(z,y)) \ for\ any\ x,y \in U.$$

Assuming T continuous, due to the associativity of max-T product, we can define for each $n \in N$ the power R^n of a fuzzy relation R recursively:

$R^1 = R$,

$R^{n+1} = R \circ R^n$ for any $n \in N$.

Definition 6. *The T-transitive closure (or T-closure R^* of a fuzzy relation R on a set U is defined by* $R^* = \sup_{n \in N} R^n$.

Proposition 5. *If R is a reflexive and symmetric fuzzy relation on a finite set U of cardinality n, then* $R^* = R^{n-1}$.

Proposition 6. *Let R be a reflexive and symmetric fuzzy relation on a set U.*

$R = R^*$ *if and only if* $T(R(x,y), R(y,z)) \leq R(x,z)$ *for any* $x, y, z \in U$.

Therefore, the transitive closure R^* of a reflexive and symmetric fuzzy relation is a similarity operator. Moreover, it is straightforward to prove that R^* is a relation greater or equal than R ($R^* \geq R$). Moreover,it can be shown [33] that if E is a similarity operator greater or equal than R, then $E \geq R^*$. In other words, the transitive closure R^* of R is the smallest similarity operator that contains R and is therefore the best upper approximation of R.

In fact, the following proposition can be proved.

Proposition 7. *Given a reflexive and symmetric fuzzy relation R on a set U and R^* its transitive closure. Let A be the set of similarity operators on U greater than or equal to R. Then*

$$R^*(x,y) = \inf_{E \in A} \{E(x,y)\}.$$

Example 4. The Min-transitive closure of R_{max} defined in Example 3 is

$R^*_{max}(x,y) = R^*_{max}(y,x)$

$R^*_{max}(x,y) = 1 \quad if\ x = y$

$R^*_{max}(M,G) = R^*_{max}(M,D) = R^*_{max}(M,W) = R^*_{max}(G,W) = R^*_{max}(G,D) = R^*_{max}(E,H) = = R^*_{max}(T,S) = R^*_{max}(W,D) = .8$

$R^*_{max}(M,C) = R^*_{max}(M,B) = R^*_{max}(M,S) = R^*_{max}(M,T) = R^*_{max}(B,S) = R^*_{max}(B,T) = = R^*_{max}(B,G) = R^*_{max}(B,W) = R^*_{max}(B,C) = R^*_{max}(B,D) = R^*_{max}(G,S) = = R^*_{max}(G,T) = R^*_{max}(G,C) = R^*_{max}(C,D) = R^*_{max}(T,D) = R^*_{max}(C,W) = = R^*_{max}(T,C) = R^*_{max}(S,W) = R^*_{max}(T,W) = R^*_{max}(S,C) = R^*_{max}(S,D) = .6$

$R^*_{max}(M,E) = R^*_{max}(M,H) = R^*_{max}(B,E) = R^*_{max}(B,H) = R^*_{max}(G,E) = R^*_{max}(G,H) = = R^*_{max}(E,C) = R^*_{max}(E,D) = R^*_{max}(E,S) = R^*_{max}(E,T) = R^*_{max}(E,W) = = R^*_{max}(H,C) = R^*_{max}(H,D) = R^*_{max}(H,S) = R^*_{max}(H,T) = R^*_{max}(H,W) = .4$

$R^*_{max}(M,P) = R^*_{max}(B,P) = R^*_{max}(G,P) = R^*_{max}(E,P) = R^*_{max}(H,P) = R^*_{max}(S,P) = = R^*_{max}(T,P) = R^*_{max}(C,P) = R^*_{max}(W,P) = R^*_{max}(D,P) = .2.$

which is a similarity relation.

3.3 Exploiting R_{min} and R_{max} in the similarity based Logic Programming

The fuzzy relations $R_{min} = \bigcap_i R_i$, and $R_{max} = \bigcup_i R_i$ are two extreme cases of aggregation rules. The first one implies that $R_{min}(a,b) \leq R_i(a,b)$ for any i, the second one that $R_{max}(a,b) \geq R_i(a,b)$ for any i. In the sequel we study properties of these relations in the framework of the Similarity-based Logic Programming. Let us start with R_{min}.

Proposition 8. *Let $R_1, \ldots, R_n$ be n similarity relations on a set U, let $\lambda \in (0,1]$ and let P be a logic program. Then,*

$$H_{\lambda,R_{\min}}(\Gamma(P)) \subseteq \bigcap_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))$$

Proof. $A \in H_{\lambda,R_{\min}}(\Gamma(P)) \Longrightarrow \exists A' \in \Gamma(P)$ t.c. $R_{\min}(A, A') \geqslant \lambda$

$\Longrightarrow \min\{R_1(A, A'),\ldots,R_n(A, A')\} \geqslant \lambda \Longrightarrow R_i(A, A') \geqslant \lambda\ \forall i = 1, \ldots, n$

$\Longrightarrow A \in H_{\lambda,R_i}(A')\ \forall i = 1, \ldots, n,\ A' \in \Gamma(P) \Longrightarrow A \in H_{\lambda,R_i}(\Gamma(P))\ \forall i = 1, \ldots, n$

$\Longrightarrow A \in \bigcap_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))$

Let us prove that the inverse inclusion does not hold.

Let the following logic program be given

$P = \{q(c)\leftarrow;\ r(c)\leftarrow;\ p(a)\leftarrow\}$

Let us suppose that R_1 and R_2 are two similarity relations defined in $L(P)$ such that

$R_1(r, q) < \lambda$, $R_1(p, q) \geqslant \lambda$, $R_1(r, p) < \lambda$;

$R_2(r, q) < \lambda$, $R_2(p, q) < \lambda$, $R_2(r, p) \geqslant \lambda$.

Then it follows:

$H_{\lambda,R_1}(\Gamma(P)) = \{q(c)\leftarrow;\ r(c)\leftarrow;\ p(a)\leftarrow; p(c)\leftarrow;\ q(a)\leftarrow\}$

$H_{\lambda,R_2}(\Gamma(P)) = \{q(c)\leftarrow;\ r(c)\leftarrow;\ p(a)\leftarrow; p(c)\leftarrow;\ r(a)\leftarrow\}$

$\bigcap_{i=1}^{2} H_{\lambda,R_i}(\Gamma(P)) = \{q(c)\leftarrow;\ r(c)\leftarrow;\ p(a)\leftarrow; p(c)\leftarrow\}$

$R_{\min}(r, q) < \lambda$, $R_{\min}(p, q) < \lambda$, $R_{\min}(r, p) < \lambda$

$H_{\lambda,R_{\min}}(\Gamma(P)) = \{q(c)\leftarrow;\ r(c)\leftarrow;\ p(a)\leftarrow\}$

Results

$\bigcap_{i=1}^{2} H_{\lambda,R_i}(\Gamma(P)) \nsubseteq H_{\lambda,R_{\min}}(\Gamma(P))$. $\diamond$

By the previous result, because the Logic Programming is a monotonic inference system, it follows that:

$$M_{H_{\lambda,R_{\min}}(\Gamma(P))} \subseteq M_{\bigcap_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))}$$

Therefore, fixed $\lambda \in (0,1]$, the extended program w.r.t. R_{min} allows to deduce less ground atomic formulae than to the intersection of the extended programs of the similarity relations $R_1, \ldots, R_n$. Moreover, we can prove the following result:

Proposition 9. *Let $R_1, \ldots, R_n$ be n similarity relations on a set U, let $\lambda \in (0,1]$ and let P be a logic program. Then,*

$$M_{\bigcap_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))} \subseteq \bigcap_{i=1}^{n} M_{H_{\lambda,R_i}(\Gamma(P))}$$

Proof. $A \in M_{\bigcap_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))} = \bigcup_{j\geqslant 0} T^j_{\bigcap_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))}(\emptyset) \Longrightarrow \exists k \geqslant 1$ t.c. $A \in T^k_{\bigcap_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))}(\emptyset)$

$\Longrightarrow \exists k \geqslant 1$ t.c. $A \in T^k_{H_{\lambda,R_i}(\Gamma(P))}(\emptyset)\ \forall i = 1, \ldots, n$

$\Longrightarrow A \in \bigcup_{j\geqslant 0} T^j_{H_{\lambda,R_i}(\Gamma(P))}(\emptyset) = M_{H_{\lambda,R_i}(\Gamma(P))}\ \forall i = 1, \ldots, n$

$\Longrightarrow A \in \bigcap_{i=1}^{n} M_{H_{\lambda,R_i}(\Gamma(P))}$

Now let us prove that the inverse inclusion does not hold.

Let the following logic program be given

$P = \{q(a)\leftarrow q(b);\ q(c)\leftarrow\}$

$M_P = \{q(c)\}$

Let us suppose that R_1 and R_2 are two similarity relations defined in $L(P)$ such that

$R_1(a,b) < \lambda$, $R_1(a,c) \geqslant \lambda$, $R_1(b,c) < \lambda$;

$R_2(a,b) < \lambda$, $R_2(a,c) < \lambda$, $R_2(b,c) \geqslant \lambda$.

Then, it results
$H_{\lambda,R_1}(\Gamma(P)) = \{q(a)\leftarrow q(b);\ q(c)\leftarrow;\ q(c)\leftarrow q(b);\ q(a)\leftarrow\}$
$M_{H_{\lambda,R_1}(\Gamma(P))} = \{q(a),\ q(c)\}$;
$H_{\lambda,R_2}(\Gamma(P)) = \{q(a)\leftarrow q(b);\ q(c)\leftarrow;\ q(a)\leftarrow q(c);\ q(b)\leftarrow\}$
$M_{H_{\lambda,R_2}(\Gamma(P))} = \{q(b),\ q(c),\ q(a)\}$;
$\bigcap_{i=1}^{2} M_{H_{\lambda,R_i}(\Gamma(P))} = \{q(a),\ q(c)\}$;
$\bigcap_{i=1}^{2} H_{\lambda,R_i}(\Gamma(P)) = \{q(a)\leftarrow q(b);\ q(c)\leftarrow\}$
$M_{\bigcap_{i=1}^{2} H_{\lambda,R_i}(\Gamma(P))} = \{q(c)\}$
Then it results
$\bigcap_{i=1}^{2} M_{H_{\lambda,R_i}(\Gamma(P))} \not\subseteq M_{\bigcap_{i=1}^{2} H_{\lambda,R_i}(\Gamma(P))}$. $\diamond$

Summarizing,
$M_{H_{\lambda,R_{\min}}(\Gamma(P))} \subseteq M_{\bigcap_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))} \subseteq \bigcap_{i=1}^{n} M_{H_{\lambda,R_i}(\Gamma(P))}$

Thus, let us define the *fuzzy least Herbrand model* of P w.r.t. the intersection relation R_{min} as:

$$M_{P,R_{min}}(A) = \sup\{\lambda \in [0,\ 1]\ /\ A \in M_{H_{\lambda,R_{min}}(\Gamma(P))}\}$$

It can be easily proved that:

Proposition 10. *Let $R_1, \ldots, R_n$ be n strict similarity relations on a set U. Then,*
$A \in M_P \iff M_{P,R_{min}}(A) = 1$

Analogous properties hold for R^*_{max}.

Proposition 11. *Let $R_1, \ldots, R_n$ be n similarity relations on a set U, let $\lambda \in (0,1]$ and let P be a logic program. Then,*
$\bigcup_{i=1}^{n} M_{H_{\lambda,R_i}(\Gamma(P))} \subseteq M_{\bigcup_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))}$

Proof. Let us consider $A \in \bigcup_{i=1}^{n} M_{H_{\lambda,R_i}(\Gamma(P))} \Longrightarrow \exists m \in \{1, \ldots, n\}$ such as $A \in M_{H_{\lambda,R_m}(\Gamma(P))} \Longrightarrow$
$\Longrightarrow A \in \bigcup_{j\geqslant 0} T^j_{H_{\lambda,R_m}(\Gamma(P))}(\emptyset) \Longrightarrow \exists k\geqslant 1$ such as $A \in T^k_{H_{\lambda,R_m}(\Gamma(P))}(\emptyset) \Longrightarrow \exists k\geqslant 1$ such as $A \in T^k_{\bigcup_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))}(\emptyset)$
$\Longrightarrow A \in \bigcup_{j\geqslant 0} T^j_{\bigcup_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))}(\emptyset) = M_{\bigcup_{i=1}^{n} H_{\lambda,R_i}(\Gamma(P))}$
Let us prove that the inverse inclusion does not hold.
Let us consider the following logic program
$P = \{p(a)\leftarrow q(c);\ r(a)\leftarrow\}$
Let us suppose that R_1 and R_2 are two similarity relations defined in $L(P)$ such that
$R_1(r, q) \geqslant \lambda$, $R_1(p, q) < \lambda$, $R_1(r, p) < \lambda$, $R_1(a, c) < \lambda$;
$R_2(r, q) < \lambda$, $R_2(p, q) < \lambda$, $R_2(r, p) < \lambda$, $R_2(a, c) \geqslant \lambda$.
Then it follows:
$H_{\lambda,R_1}(\Gamma(P)) = \{p(a)\leftarrow q(c);\ r(a)\leftarrow;\ q(a)\leftarrow;\ p(a)\leftarrow r(c)\}$
$M_{H_{\lambda,R_1}(\Gamma(P))} = \{q(a),\ r(a)\}$;
$H_{\lambda,R_2}(\Gamma(P)) = \{p(a)\leftarrow q(c);\ r(a)\leftarrow;\ p(c)\leftarrow q(c);\ p(a)\leftarrow q(a);\ r(c)\leftarrow\}$
$M_{H_{\lambda,R_2}(\Gamma(P))} = \{r(c),\ r(a)\}$;
$\bigcup_{i=1}^{n} M_{H_{\lambda,R_i}(\Gamma(P))} = \{r(c),\ r(a),\ q(a)\}$;

$\bigcup_{i=1}^{2}$ $H_{\lambda,R_i}(\mathit{\Gamma}(P))$ = {p(a)←q(c); r(a)←; q(a)←; p(a)←r(c); p(c)←q(c)←; p(a)←q(a); r(c)←}
$H_{\lambda,R_i}(\mathit{\Gamma}(P))$ = {q(a)←q(b); q(c)←}
$M_{\bigcup_{i=1}^{2} H_{\lambda,R_i}(\mathit{\Gamma}(P))}$ = {r(c), r(a), q(a), p(a)}
Then, it follows:
$M_{\bigcup_{i=1}^{2} H_{\lambda,R_i}(\mathit{\Gamma}(P))} \nsubseteq \bigcup_{i=1}^{2} M_{H_{\lambda,R_i}(\mathit{\Gamma}(P))}$. ⋄

Furthermore, it results that

Proposition 12. *Let $R_1, \ldots, R_n$ be n similarity relations on a set U, let $\lambda \in (0,1]$ and let P be a logic program. Then,*
$\bigcup_{i=1}^{n} H_{\lambda,R_i}(\mathit{\Gamma}(P)) \subseteq H_{\lambda,R^*_{\max}}(\mathit{\Gamma}(P))$

Proof. $C \in \bigcup_{i=1}^{n} H_{\lambda,R_i}(\mathit{\Gamma}(P)) \Longrightarrow \exists j \in \{1, \ldots, n\}$ t.c. $C \in H_{\lambda,R_j}(\mathit{\Gamma}(P)) \Longrightarrow$

$\Longrightarrow \exists C' \in \mathit{\Gamma}(P)$ t.c. $R_j(C, C') \geqslant \lambda \Longrightarrow R_{\max}(C, C') \geqslant \lambda \Longrightarrow R^*_{\max}(C, C') \geqslant \lambda \Longrightarrow$
$\Longrightarrow C \in H_{\lambda,R^*_{\max}}(\mathit{\Gamma}(P))$
Let us prove that the inverse inclusion does not hold.
Let us consider two similarity relations R_1 ed R_2 defined in a first order languages L such that
$R_1(a, b) = 0.3$, $R_1(a, c) = 0.3$, $R_1(b, c) = 0.5$
$R_2(a, b) = 0.6$, $R_2(a, c) = 0.4$, $R_2(b, c) = 0.4$
then
$R_{\max}(a, b) = 0.6$, $R_{\max}(a, c) = 0.4$, $R_{\max}(b, c) = 0.5$
and
$R^*_{\max}(a, b) = 0.6$, $R^*_{\max}(a, c) = 0.5$, $R^*_{\max}(b, c) = 0.5$
Let P = {a←}
Then, results that
$H_{0.5,R_1}(\mathit{\Gamma}(P))$ = {a←}
$H_{0.5,R_2}(\mathit{\Gamma}(P))$ = {a←; b←}
$\bigcup_{i=1}^{2} H_{0.5,R_i}(\mathit{\Gamma}(P))$ = {a←; b←}
$H_{0.5,R^*_{\max}}(\mathit{\Gamma}(P))$ = {a←; b←; c←}
Then,
$H_{0.5,R^*_{\max}}(\mathit{\Gamma}(P)) \nsubseteq \bigcup_{i=1}^{2} H_{0.5,R_i}(\mathit{\Gamma}(P))$ ⋄

Then, because the Logic Programming is a monotonic inference system
$M_{\bigcup_{i=1}^{n} H_{\lambda,R_i}(\mathit{\Gamma}(P))} \subseteq M_{H_{\lambda,R^*_{\max}}(\mathit{\Gamma}(P))}$
Therefore, fixed $\lambda \in (0,1]$, the extended program w.r.t. R^*_{max} allows to deduce more ground atomic formulae than to the union of the extended programs of the similarity relations $R_1, \ldots, R_n$.

Summarizing,
$\bigcup_{i=1}^{n} M_{H_{\lambda,R_i}(\mathit{\Gamma}(P))} \subseteq M_{\bigcup_{i=1}^{n} H_{\lambda,R_i}(\mathit{\Gamma}(P))} \subseteq M_{H_{\lambda,R^*_{\max}}(\mathit{\Gamma}(P))}$

Let us define the *fuzzy least Herbrand model* of P w.r.t. the union relation R^*_{max} as:

$$M_{P,R^*_{max}}(A) = \sup\{\lambda \in [0, 1] \ / \ A \in M_{H_{\lambda,R^*_{max}}(\mathit{\Gamma}(P))}\}$$

It can be proved that:

Proposition 13. *Let $R_1, \ldots, R_n$ be n strict similarity relations on a set U and let P be a logic program. Then,*

$$A \in M_P \iff M_{P,R^*_{max}}(A) = 1$$

4 Conclusion

In this paper we studied two estreme cases (R_{min} and R_{max}) of group preferences relationsimposing the min-transitivity property. The operators Min and Max (intersection and union in terms of fuzzy relations) are two extreme cases of aggregation rules. Properties of these relations in the framework of Similarity-based Logic Programming have been proved. As future work different aggregation rules will be studied in order to manage the similarity values in a less extreme way.

Acknowledgements

The authors gratefully acknowledge the referee for their useful comments which have contributed to improve this work.

References

1. Apt R.K., *Logic Programming*, in: J. van Leeuwen (Ed.), Handbook of Theoretical Computer Science, vol. B, (Elsevier, Amsterdam, 1990) 492-574.
2. Bezdek JC, Harris JO. *Fuzzy partitions and relations: An axiomatic basis for clustering.* Fuzzy Sets Syst 1978;1:112127.
3. Bodenhofer U. *A Similarity-Based Generalization of Fuzzy Orderings.* (1999) Vol. C 26 of Schriftenreihe der Johannes-Kepler-Universitat Linz, Universitatsverlag Rudolf Trauner.
4. Dubois D., Prade H., *Resolution principles in possibilistic logic*, Int. Journal of Approximate Reasoning, 3 (1990) 1-21.
5. J.C. Fodor and S. Ovchinnikov, *On aggregation of T-transitive fuzzy binary relations.* Fuzzy Sets and Systems 72 (1995) 135-145. (MR 1335529)
6. Fodor, J., Roubens, M. *Fuzzy preference modelling and multicriteria decision support.* Kluwer Ac. Publ., 250 pages, ISBN 0-7923-3116-8, 1994.
7. Genesereth M.R., Ketchpel S.P. *Software agents.* Communications of the ACM, 37(7):48-53, 1994.
8. Gerla G., Sessa M.I., *Similarity in Logic Programming*, in: G. Chen, M. Ying, K.-Y. Cai (Ed.s), Fuzzy Logic and Soft Computing, (Kluwer Acc. Pub., Norwell, 1999), 19-31.
9. Hajek P. *Metamathematics of fuzzy logic.* Dordrecht, The Netherlands: Kluwer; 1998.
10. Ishizuka M., Kanai N., *PROLOG-Elf incorporating fuzzy logic*, Proc. 9th Int. Joint. Conf. on Artificial Intelligence (Springer, Berlin, 1985) 701-703.
11. Joan Jacas, Jordi Recasens: Aggregation of T-transitive relations. Int. J. Intell. Syst. 18(12): 1193-1214 (2003)
12. Kifer M., Li A., *On the Semantic of Rule-Baaed Expert Systems with Uncertainty*, Inter. Conf. on Databases Theory 1988, Lecture Notes in Computer Science, vol. S26 (Springer, Berlin, 1988) 186-202.
13. Kifer M., Subrahmanian V.S., *Theory of Generalized Annotated Logic Programming and its Applications*, Journal of Logic, Programming 12(3&4) (1992) 335-367.
14. Kowalski, R.A., *Predicate logic as a programming language*, in: Proc. IFIP'74 (1974) 569-574.
15. Formato F., Gerla G., Sessa M.I., *Similarity-based unification*, Fundamenta Informaticae 40 (2000) 1-22.
16. Loia V., Luongo P., Senatore S. and Sessa M.I., *A Similarity-based View to Distributed Information Retrieval with Mobile Agents.* In The 10th IEInternational Conference on Fuzzy Systems, Melbourne, Australia, December 2-5, 2001.

17. Loia V., Senatore S. and Sessa M.I., *Similarity-based agents for email mining.* In Proc. of the joint Conference IFSA/NAFIPS 2001, Vancouver, Canada, July 25-28, 2001.
18. Loia V., Senatore S. and Sessa M.I., *Similarity-based SLD Resolution and its implementation in an Extended Prolog System*, Proceedings of 10th IEEE International Conference on Fuzzy Systems, Melbourne, Australia, December 2-5 2001. IEEE PRESS.
19. Martin T.P., Baldwin J.F., Pilsworth B.W., *The implementation of FPROLOG a fuzzy PROLOG interpreter*, Fuzzy Sets and Systems 23 (1987) 119-129.
20. Mukaidono M., Shen Z.L., Ding L., *Fundamentals of fuzzy PROLOG*, Int. Journal of Approximate Reasoning 3 (1989) 179-193.
21. Ovchinnikov S.V. *Aggregating transitive fuzzy binary relations.* IJUFKBS 1995;3:4755.
22. Ovchinnikov S.V. *Similarity relations, fuzzy partitions, and fuzzy orderings.* Fuzzy Sets and Systems, (1991) 40, 107-126.
23. Ovchinnikov S.V. *Transitive fuzzy orderings of fuzzy numbers, Fuzzy Sets and Systems.* V.30 n.3, p.283-295, May 10, 1989
24. Ovchinnikov, S. V. *Representations of transitive fuzzy relations.* In: Skala, Termini, and Trillas, 1983;105-118.
25. Poincaré J.H. *La Valeur de la Science.* Flammarion, Paris (1905).
26. Pradera A, Trillas E, Castineira E. *On the aggregation of some classes of fuzzy relations.* In: Bouchon-Meunier B, Gutierrez J, Magdalena L, Yager R, editors. Technologies for constructing intelligent systems. Heidelberg: Springer-Verlag; 2002. pp 125147.
27. Pradera A, Trillas E. *A note on pseudometrics aggregation.* Int J Gen Syst. 2002;31(1):4151.
28. Saminger S., Mesiar R., Bodenhofer U. *Domination of Aggregation Operators and Preservation of Transitivity.* International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems 10(Supplement): 11-36 (2002)
29. Sessa M.I., *Approximate Reasoning by Similarity-based SLD Resolution*, Theoretical Computer Science, 275 (2002) 389-426.
30. Sessa M.I. *Translations and Similarity-based Logic Programming*, Soft Computing 5(2) (2001).
31. Subrahmanian V.S. *On the semantics of quantitative logic*, Proc. IEEE Symposium on Logic Programming (1987) 173-182.
32. Trillas E., Valverde L. *An Inquiry into Indistinguishability Operators.* In: H.J. Skala, S. Termini, E. Trillas (eds.), Aspects of Vagueness. Reidel, Dordrecht (1984), 231-256.
33. Valverde L. *On the structure of F-indistinguishability operators.* Fuzzy Sets Syst 1985;17:313328.
34. Zadeh, L.A., *Similarity Relations and Fuzzy Orderings.* Information Sciences 3 (1971) 177-200.

A declarative approach to uncertainty orders*

Andrea Capotorti[1] and Andrea Formisano[2]

[1] Dipartimento di Matematica e Informatica, Università di Perugia.
capot@dipmat.unipg.it
[2] Dipartimento di Informatica, Università dell'Aquila.
formisano@di.univaq.it

Abstract. Traditionally, most of the proposed probabilistic models of decision under uncertainty rely on numerical measures and representations. Alternative proposals call for qualitative (non-numerical) treatment of uncertainty, based on preference relations and belief orders.
The automation of both numerical and non-numerical frameworks surely represents a preliminary step in the development of inference engines of intelligent agents, expert systems, and decision-support tools.
In this paper we exploit Answer Set Programming to formalize and reason about uncertainty expressed by belief orders. The availability of ASP-solvers supports the design of automated tools to handle such formalizations. Our proposal reveals particularly suitable whenever the domain of discernment is *partial*, i.e. it does not represent a closed world but just the relevant part of a problem.
We first illustrate how to automatically "classify", according to the most well-known uncertainty frameworks, any given partial qualitative uncertainty assessment. Then, we show how to compute the enlargement of an assessment to any other new inference target, with respect to a fixed (admissible) qualitative framework.

Key words: Uncertainty orders, answer set programming, partial assessments, general inference.

Probability does not exist!
—Bruno de Finetti [13]

Introduction and background

Nowadays, several numerical tools are usually adopted in AI to represent and manage uncertainty. All of them originate from amendments of the well-known Probability measure, aimed at generalizing it to better fit different peculiarities of specific application fields (for a survey the reader can refer to [21, 30] or to [25, Chapters 8–10], among others. The Appendix briefly summarizes some basic notions about uncertainty measures, from the quantitative point of view). The measures that achieved wider diffusion can be classified as:

* Research partially funded by the *Information Society Technologies programme of the European Commission, Future and Emerging Technologies* under the IST-2001-37004 WASP project.

- Capacities;
- Possibility and Necessity measures;
- Probabilities;
- Belief and Plausibility functions;
- Lower and Upper probabilities.

Among all these measures (which are real-valued functions), Capacities [7] characterize the weakest notion. Indeed, Capacities are measures whose unique property is monotonicity with respect to the implication of events. Namely, if a situation A implies a situation B, then the uncertainty on A should be not greater than the uncertainty on B. Uncertainty models based on such measures are very general but, on the other hand, very weak because they describe nothing more than "common sense" behaviors. The class of Capacities includes all other classes.

Possibility measures (Necessity measures are their dual, cf. Def. 2 of the Appendix) come from *Fuzzy theory* [17, 33] and originate from the need to express "vagueness" about the descriptions of situations instead of uncertainty about their truth.

Probabilities are characterized by the "additivity" property: Having judged the uncertainties $P(A)$ and $P(B)$ on any pair of disjoint situations A and B, the uncertainty on their combination $A \vee B$ is defined as $P(A) + P(B)$. Such measures have a wide range of applications. Almost any medical, engineering, economic, and environmental decision-aid tool is usually built on (or at least compared to) probabilistic models.

Belief functions (whose dual are Plausibilities) are the base of *Evidence theory* [26]. The Belief on a proposition represents the "strength" by which a not fully detailed information supports its truth. Plausibility functions, on the other hand, represent how much the evidence makes reasonable that a proposition is true. Such uncertainty measures have found valuable application in economic and medical frameworks where the initial available information is quite not-specific.

Lower probabilities (whose dual are Upper probabilities) are instead adopted whenever one needs to consider as valid an entire family of probabilistic models in place of a single one. Such measures have been developed within the field of *Imprecise probabilities* [11, 29]. Obviously, such uncertainty measures are usually adopted in each context where precise probabilities are typically used, but where there are not enough constraints to be obliged to use a unique model.

As a matter of fact, each one of the framework described so far, can manage uncertainty and retains all of the expressive power of mathematical quantitative models. Though, inevitably, they suffer from the drawbacks often faced whenever numerical models are applied to practical problems: *a)* the difficulty of expressing a complete evaluation, and *b)* the hardness to elicit precise numerical values. The former problem can be circumvented by following the pioneering approach proposed by de Finetti in the context of Probabilities [12, 13]. Namely, by introducing the so called *partial models*, i.e. numerical evaluations defined only on some of the situations at hand, and intended to be a restriction of some of the complete models mentioned above. (Then, we will deal with partial Capacities,

partial Probabilities, and so on.) This approach allows the analyst of the problem to focus his/her evaluation on the situations really judged relevant, w.r.t. the problem at hand. This leaves open the possibility to enlarge the model to other scenarios that could enter on the scene later. To obviate the latter drawback of numerical models, *qualitative approaches* have been proposed in the last decades. The central idea of such methodologies is to grade uncertainty about the truth of propositions, through comparisons expressing the judgement of "less or more believed to be true". This operationally translates into the use of (partial) order relations in place of numerical grades.

Qualitative approaches are receiving wider and wider attention, either as theoretical tools to deal directly with belief management [3, 10, 14], or inside the more articulated framework of decision-making theory (see, for example, [15, 16, 18, 20, 22]). This is because, they better fit the nature of human judgments.

Numerical models remain anyway a reference point. Both because their properties are well-known and deeply investigated, and because, when profitably involved, they could bring to conclusions hardly achievable by purely qualitative tools. The connection between the qualitative and the numerical frameworks is usually expressed by the requirement that the qualitative order must be representable[3] by a (partial) numerical model. Representability of an order guarantees that the comparisons among the propositions follow the same rationale of the kind of numerical model agreeing with. Hence, the basic properties of the way in which different pieces of information are combined is maintained.

In the next section we show that representability of orders, defined on arbitrary finite sets of propositions, can be characterized by the specific properties (axioms). Before to enter into such details, it is worth stressing that in this paper we adopt an alternative approach, by inverting the usual attitude towards qualitative management of uncertainty. In fact, specific axioms are usually set in advance, so that only order relations satisfying them are admitted. Here, on the contrary, given a fixed preference relation (for instance, directly issuing from analyst's interpretation of real world), our goal consists in ascertain what are the reasonable rules to work with. This will be made easy thanks to the expressive power of Answer Set Programming [23, 24]. In fact, most of such axioms are of direct declarative reading, as they involve only logical and preference relations. As we will see, such a declarative character supports a straightforward translation of the axioms within the logical framework of Answer Set Programming. As a consequence, we immediately obtain an executable specification able to discriminate between the different uncertainty orders. More specifically, we exploit a solver (in our case smodels, cf. [1]) to determine the set of axioms that are violated by a given preference relation, which expresses user's beliefs comparisons.

Then, we move the first step toward the implementation of an inference engine that borrows user's conceptualization of uncertainty and (implicitly) adopts

[3] Recall that, in general, a numerical assessment f on a set of propositions $A_1, \ldots, A_n$ represents (or, equivalently, induces) a qualitative order $\preceq^*$ among them if, for each pair A_i, A_j it holds that $A_i \preceq^* A_j \Longleftrightarrow f(A_i) \leqslant f(A_j)$.

his/her own way of modeling the intrinsic properties of the problem at hand. Thus, the system tries to mimic user's way of expressing lack of information and variability of phenomena. By acting in this manner, once the (most specific) framework closest to user's modelization is detected, it can be used to infer reasonable conclusions about proposition not comprised in the initial domain. This process is usually referred to as *order extension*. The availability of order-extension techniques is one of the main advantages offered by the use of partial models in the treatment of uncertainty.

The paper is organized as follows. Next section briefly describes the axioms characterizing partial uncertainty relations (notice that we focus on the treatment of partial orders, even if total relations can easily be dealt with by exploiting the very same machinery). Sec. 2 recalls the main features of Answer Set Programming, with particular emphasis on the application to the above mentioned issues. In Sections 3 and 4 we illustrate, also by simple examples, the potentialities of our approach. Finally, we draw conclusions and outline future developments.

1 Characterization of uncertainty orders

When one admits that nothing is certain one must, I think, also add that some things are more nearly certain than others.

—Bertrand Russell

By following the way paved by [8, 14, 31, 32], various (qualitative) preference orders have been fully classified in [4, 5, 6] according to their agreement with the most well-known numerical models; both for complete and partial assessments.

In particular, apart from Possibility and Necessity measures—that seem to have an intrinsically numerical character— [6] proposes a fully axiomatic classification of partial orders according to the numerical models outlined above.

Let us start by briefly recalling the basic notions on uncertainty orders and their axiomatic characterization. We will not enter into the details of the motivations for such classification, the reader is referred to [4, 5, 6]. The domain of discernment is represented by a finite set of events $\mathcal{E} = \{E_1, \ldots, E_n\}$ (among them, $\emptyset$ and Ω denote the impossible and the sure event, respectively). The events in $\mathcal{E}$ are seen as the relevant propositions on which the subject of the analysis can (or wants) to express his/her opinion. Hence, usually $\mathcal{E}$ does not represent a full model, i.e. it does not comprehend all elementary situations and all of their combinations. For this reason, a crucial component of partial assessments is the knowledge of the logical relationships (incompatibilities, implications, combinations, equivalences, etc.) holding among the events E_is. Such constraints are usually represented as a set $\mathcal{C}$ of clauses predicating on the E_is.

Taking into account the constraints $\mathcal{C}$, the family $\mathcal{E}$ spans a minimal Boolean algebra $\mathcal{A}_{\mathcal{E}}$ containing $\mathcal{E}$ itself. Note that $\mathcal{A}_{\mathcal{E}}$ is only implicitly defined via $\mathcal{E}$ and $\mathcal{C}$ and it is not a part of the assessment. Anyway, $\mathcal{A}_{\mathcal{E}}$ can be referenced as a supporting structure.

Let $\preceq$ be a partial (i.e. not necessarily defined for all pairs (A, B) in $\mathcal{E} \times \mathcal{E}$) order among events, expressing the intuitive idea of being "less or equal than" or "not preferred to". The symbols $\sim$ and $\prec$ denote the symmetrical part and asymmetrical part of $\preceq$, respectively.

As mentioned before, Capacities constitute the most general numerical tool to manage uncertainty and they express "common sense" behaviors. Hence, in our context, any reasonable relation $\preceq$ must be representable by a partial Capacity (i.e., a restriction to the events under consideration, of a Capacity measure). This translates into the following axioms: the (partial) order $\preceq$ must be a reflexive binary relation on $\mathcal{E}$ such that

(A1) $\prec$ has no intransitive cycles;[4]

(A2) $\neg(\Omega \preceq \phi)$;

(A3) for all $A, B \in \mathcal{E}$, $A \subseteq B \Longrightarrow \neg(B \prec A)$;

where $\neg(B \prec A)$ means that the pair (B, A) does not belong to $\prec$.
Mathematical properties of orders satisfying basic axioms (A1), (A2) and (A3) are deeply investigated in [10]. In what follows, we consider these axioms as prerequisites for any investigation on $\preceq$. Differentiation among order relations can be done on the basis of more specific way of combining distinct pieces of information. Below, we list the axioms characterizing each class.[5] The name of the classes comes from the representability of $\preceq$ by corresponding partial numerical measures.[6]

Comparative Probabilities. An order $\preceq$ is *representable by a partial Probability assessment* iff the following holds:

(CP) for any $A_1, \ldots, A_n, B_1, \ldots, B_n \in \mathcal{E}$, with $B_i \preceq A_i$, $\forall i = 1, \ldots, n$, such that for some $r_1, \ldots, r_n > 0$, if $\sup \sum_{i=1}^{n} r_i(a_i - b_i) \leqslant 0$ holds than, for all $i = 1, \ldots, n$, $A_i \sim B_i$ (a_i, b_i denote the indicator functions of A_i, B_i, resp.).

Comparative Beliefs. An order $\preceq$ is *representable by a partial Belief function assessment* iff for all $A, B, C \in \mathcal{E}$ s.t. $A \subset B$, $B \wedge C = \phi$ it holds that

(B) $\quad A \prec B \Longrightarrow \neg(B \vee C \preceq A \vee C)$.

Comparative Lower probabilities. An order $\preceq$ is *representable by a partial Lower probability assessment* iff for all $A, B \in \mathcal{E}$ s.t. $A \wedge B = \phi$ it holds that

(L) $\quad \phi \prec A \Longrightarrow \neg(A \vee B \preceq B)$.

---

[4] A preference relation $\prec$ on a set X has an intransitive cycle if there exist $A_1, \ldots, A_n \in X$ for $n > 2$ such that $A_i \prec A_{i+1}$ holds for each $i = 1, \ldots, n-1$, while $A_1 \prec A_n$ does not hold.

[5] Note that we characterize each class by a single axiom, whereas in [6] some classes are described by introducing further axioms. It is easy to see that these additional axioms are redundant whenever we consider to enlarge $\prec$ by monotonicity (i.e. by imposing that $A \subseteq B \Longleftrightarrow A \prec B$ always holds).

[6] Axiom (CP) was originally introduced in [8]. Axiom (B) derives by the analogous axiom introduced for complete orders in [32].

Comparative Plausibilities. An order $\preceq$ is *representable by a partial Plausibility function assessment* iff for all $A, B, C \in \mathcal{E}$ s.t. $A \subset B$ it holds that

(PL) $\quad A \sim B \Longrightarrow \neg(A \vee C \prec B \vee C)$.

Comparative Upper probabilities. An order $\preceq$ is *representable by a partial Upper-probability assessment* iff for all $A, B, C \in \mathcal{E}$ s.t. $A \wedge B = \phi$ it holds that

(U) $\quad \phi \sim A \Longrightarrow \neg(C \prec A \vee C)$.

Comparative Lower/Upper probabilities. An order $\preceq$ can be simultaneously *represented by both a partial Lower-probability assessment* and *by a partial Upper-probability assessment* iff it simultaneously satisfies both axioms (L) and (U).

Note that only the axiom (CP) does not have a pure qualitative nature since it involves indicator functions and summations. Such axiom is the only one whose verification should require some form of numerical elaboration (e.g. involving some linear programming tool such as the simplex or the interior point methods). Meanwhile, to remain within the same kind of axioms, the following *necessary* axiom (WC) can also be considered. Note that (WC), if taken by itself, does not guarantee the representability of $\preceq$ by a partial Probability assessment; nevertheless, its failure witnesses non-representability.

***Weak* comparative probabilities.** If $\preceq$ is *representable by a partial Probability assessment* then, for all $A, B, C \in \mathcal{E}$ s.t. $A \wedge C = B \wedge C = \phi$ it holds that

(WC) $\quad A \preceq B \Longrightarrow \neg(B \vee C \prec A \vee C)$

Clearly, all such qualitative axioms are of direct reading, i.e. they explicit which are the rules to follow in combining elements of the domain $\mathcal{E}$ to remain inside a specific framework.

The introduction of different classes of orders shares the very same motivations supporting the definition of different numerical measures of uncertainty. The main point is that there exist practical situations where a strictly probabilistic approach is not viable. The following example describes an extremely simplified situation of this kind.

Example 1. Let A, B, and C be three distinct companies, and let each of them be a potential buyer of a firm that some other company wants to sell. Even being distinct, both A and C belong to the same holding. Hence, the following uncertainty order about which company will be the buyer, could reflect specific information about the companies' strategies (by abuse of notation, let A denote the event "the company A buys the firm", and similarly for B and C):

$$\emptyset \prec A \prec B \prec B \vee C \prec A \vee C \prec \Omega.$$

Since A, B and C are incompatible events, it is immediate to see that the order relation is not representable by a probability because it violates axiom (WC), while it can be managed in line with Belief functions behaviors because it agrees with axiom (B).

2 Answer set programming

In the following sections we show how to obtain executable specifications from the axiomatic classification of preference orders described so far. To this end, we employ Answer Set Programming (ASP, for short).

Let us first briefly recall the basics of such alternative style of logic programming [23, 24]. A problem can be encoded—by using a function-free logic language—as a set of properties and constraints which describe the (candidate) solutions. More specifically, an *ASP-program* is a collection of *rules* of the form

$$L_1; \ldots; L_k; not\, L_{k+1}; \ldots; not\, L_\ell \;\leftarrow\; L_{\ell+1}, \ldots, L_m, not\, L_{m+1}, \ldots, not\, L_n$$

where $n \geqslant m \geqslant \ell \geqslant k \geqslant 0$ and each L_i is a literal, i.e., an atom A or a negation of an atom $\neg A$. The symbol $\neg$ denotes classical negation, while *not* stands for negation-as-failure (Notice that $','$ and $';'$ stand for logical conjunction and disjunction, respectively.) The left-hand side and the right-hand side of the clause are said *head* and *body*, respectively. A rule with empty head is a *constraint*. Intuitively, the literals in the body of a constraint cannot be all true, otherwise they would imply falsity.

Semantics of ASP is expressed in terms of *answer sets* (or equivalently *stable models*, cf. [19]). Consider first the case of an ASP-program P which does not involve negation-as-failure (i.e., $\ell = k$ and $n = m$). In this case, a set X of literals is said to be closed under P if for each rule in P, whenever $\{L_{\ell+1}, \ldots, L_m\} \subseteq X$, it holds that $\{L_1, \ldots, L_k\} \cap X \neq \emptyset$. If X is inclusion-minimal among the sets closed under P, then it is said to be an answer set for P. Such a definition is extended to any program P containing negation-as-failure by considering the *reduct* P^X (of P). P^X is defined as the set of rules

$$L_1; \ldots; L_k \;\leftarrow\; L_{\ell+1}, \ldots, L_m$$

for all rules of P such that X contains all the literals $L_{k+1}, \ldots, L_\ell$, but does not contain any of the literals $L_{m+1}, \ldots, L_n$. Clearly, P^X does not involve negation-as-failure. The set X is an answer set for P if it is an answer set for P^X.

Once a problem is described as an ASP-program P, its solutions (if any) are represented by the answer sets of P. Notice that an ASP-program may have none, one, or several answer sets.

Let us consider the program P consisting of the two rules

$$p; q \leftarrow \qquad\qquad \neg r \leftarrow p.$$

Such a program has two answer sets: $\{p, \neg r\}$ and $\{q\}$. If we add the rule (actually, a constraint) $\leftarrow q$ to P, then we rule-out the second of those answer sets, because it violates the constraint. This simple example reveals the core of the usual approach followed in formalizing/solving a problem with ASP. Intuitively speaking, the programmer adopts a "generate-and-test" strategy: first (s)he provides a set of rules describing the collection of (all) potential solutions. Then, the addition of a group of constraints rules-out all those answer sets that are not desired real solutions.

To find the solutions of an ASP-program, an ASP-solver is used. Several solvers have became available (cf. [1], for instance), each of them being characterized by its own prominent valuable features.

Expressive power of ASP, as well as, its computational complexity have been deeply investigated. The interested reader can refer to the survey [9], among others, for a comparison of expressive power and computational complexity of various forms of logic programming.

As we will see, in this work we choose smodels as solver, together with its natural front-end lparse [28].

Let us give a simple example of ASP-program (see [2], among others, for a presentation of ASP as a tool for declarative problem-solving). In doing this, we will recall the syntax of smodels as well as the main features of lparse/smodels which will be exploited in the rest of the paper (see [28], for a much detailed description). The problem we want to formalize in ASP is the well-known n-*queens* problem: "Given a $n \times n$ chess board, place n queens in such a way that no two of them attack each other". The clauses below state that a candidate solution is any disposition of the queens, provided that each column of the board contains one and only one queen. (The fact that a queen is placed on the nth column and on the mth row is encoded by the atom queen(n,m).)[7]

```
position(1..n).
1{queen(Col,Row) : position(Col)}1 :- position(Row).
```

The second rule is a particular form of constraint available in smodels' language. The general form of such a kind of clauses is

$$k\{\langle property_def\rangle : \langle range_def\rangle\}m \text{ :-} \langle search_space\rangle$$

where: the conditions $\langle search_space\rangle$ in the body define the set of objects of the domain to be checked; the atom $\langle property_def\rangle$ in the head defines the property to be checked; the conjunction $\langle range_def\rangle$ defines the possible values that the property may take on the objects defined in the body, namely by providing a conjunction of unary predicates each of them defining a range for one of the variables that occur in $\langle property_def\rangle$ but not in $\langle search_space\rangle$; k and m are the minimum and maximum number of values that the specified property may take on the specified objects. (Notice that this form of constraint, available in smodels, actually is syntactic sugar, since it can be translated into "proper" ASP-clauses thanks to negation, cf. [28, 27].)

We now introduce two constraints, in order to rule out those placements where two queens control either the same row or the same diagonal of the board:

```
:- queen(Col,Row1), queen(Col,Row2),
   position(Col), position(Row1), position(Row2),
   Row1 < Row2.
:- queen(Col1,Row1), queen(Col2,Row2),
   position(Col1), position(Col2), position(Row1), position(Row2),
   Row1 < Row2, abs(Col1-Col2) == abs(Row1-Row2).
```

Here is some of the answer sets produced by smodels, when fed with our program (together with a value for the constant n, in this case we put n= 8).

[7] In the syntax of smodels ':-' denotes implication $\leftarrow$, while ',' stands for conjunction. Moreover, the constant n occurring in the first clause, can be seen as a parameter of the program, supplied to the solver at run-time.

```
Answer: 1.
Stable Model: queen(4,1) queen(6,2) queen(1,3) queen(5,4) queen(2,5)
  queen(8,6) queen(3,7) queen(7,8) ...
Answer: 2.
Stable Model: queen(4,1) queen(2,2) queen(8,3) queen(5,4) queen(7,5)
  queen(1,6) queen(3,7) queen(6,8) ...
...
```

Notice that lparse offers some elementary built-in arithmetic functions (such as abs(), in the above clause) that can be used to perform simple arithmetics. More in general, lparse allows the user to employ user-defined C or C++ functions within an ASP-program. The object code of these functions needs only to be linked with lparse at run time. (The interested reader is referred to [28] for a detailed description of this feature.) We exploited this feature (not directly available in some other solvers) to implement a basic library of functions aimed at handling sets and operation on sets.

The pair lparse/smodels constitutes an essential and neat tool for fast prototypical development. Moreover notable facilities come from the simple albeit useful capability of integration with the C programming language, the prompt availability of the source-code (under the GNU General Public License) and documentation, and the ease of use.

3 Preference classification

Our first task consists in writing an ASP-program able to classify any given partial order $\preceq$, w.r.t. the axioms seen in Sec. 1 (except for (CP), that, up to our knowledge, does not admit a purely declarative formulation). A preliminary step is the introduction of suitable predicates, namely, prec(·,·), precneq(·,·), and equiv(·,·), to render in ASP the relators $\preceq$, $\prec$, and $\sim$, respectively. Moreover, the fact of "being an event" (i.e. a member of $\mathcal{E}$) is stated through the monadic predicate event(·).[8] Auxiliary predicates/functions are defined to render usual set-theoretical constructors, such as $\cap$, $\cup$, and $\subseteq$, which, as mentioned, have been made available by linking user-defined C-libraries.

The characterization of potential legal answer sets is done by asserting properties of prec(·,·), precneq(·,·), and equiv(·,·), by means of the following rules:

```
prec(E1,E2) :- event(E1), event(E2), equiv(E1,E2).
prec(E2,E1) :- event(E1), event(E2), equiv(E1,E2).
equiv(E1,E2) :- event(E1), event(E2), prec(E2,E1), prec(E1,E2).
prec(E1,E2) :- event(E1), event(E2), precneq(E1,E2).
:- precneq(E1,E2), event(E1), event(E2), equiv(E1,E2).
```

Also axioms (A1), (A2), and (A3) must be imposed. For instance (A3) is rendered by:

```
:- event(E1), event(E2), subset(E1,E2), precneq(E2,E1).
```

[8] Actually, in our program, events are denoted by integer numbers. Here, for the sake of readability, we systematically denote events by capital letters.

This rules-out all answer sets in which there exist two events E_1 and E_2 such that both $E_1 \subseteq E_2$ and $E_2 \prec E_1$ hold.

Consider now one of the axioms of Sec. 1, say (B), for simplicity. Since, in this phase, we do not want to impose such axiom, but we just want to test whether or not it is satisfied by the preference relation at hand, we introduce a rule of the form:

```
failsB :- event(A), event(B), event(C), subset(A,B), A!=B, empty(interset(B,C)),
          precneq(A,B), prec(unionset(B,C),unionset(A,C)).
```

whose meaning is that the fact failsB is true (i.e. belongs to the answer set) whenever there exist events falsifying axiom (B). Having in mind the axiom (B) of Sec. 1, this clause is of immediate reading. Analogous treatment has been done for all other axioms (L), (U), (PL), and (WC).

When smodels is fed with such program, together with a description of an input preference relation (i.e., a collection of facts of the forms prec(·,·), precneq(·,·), and equiv(·,·)), different outcomes may be obtained:

a) If no answer set is produced, then the input preference relation violates some basic requirement, such as axioms (A1), (A2), or (A3).

b) Otherwise, if an answer set is generated, there exists a numerical (partial) model representing the input preference order. Moreover, the presence in the answer set of a fact of the form fails$\mathcal{C}$ (say failsL, for example), witnesses that the corresponding axiom ((L) in the case) is violated by the given preference order. Consequently, the given order (as well as its extensions) is not compatible with the uncertainty framework ruled by $\mathcal{C}$ (in the case of failsL, the given order cannot be represented by a partial Lower probability).

Example 2. Suppose a physician wants to perform a preliminary evaluation about the reliability of a test for SARS (Severe Acute Respiratory Syndrome). Up to his/her knowledge, the SARS diagnosis is based on moderate or severe respiratory symptoms and on the positivity or indeterminacy of an adopted clinical test about the presence of the SARS-associated antibody coronavirus (SARS-CoV). The elements appearing in his/her analysis can be schematized as:

A $\equiv$ *Normal respiratory symptoms*
B $\equiv$ *Moderate respiratory symptoms*
C $\equiv$ *Severe respiratory symptoms*
D $\equiv$ *Moderate or sever respiratory symptoms*
E $\equiv$ *Death from pulmonary diseases*
F $\equiv$ *Positive or indeterminate clinical test*

subject to these (logical) restrictions:

$$A \cap B = \emptyset,\ B \cap C = \emptyset,\ A \cap C = \emptyset,\ A \cup B \cup C = \Omega,\ D = A \cup B,\ E \subset C,\ F \cap A = \emptyset.$$

Consider the following partial order:

```
precneq(∅,C).   precneq(C,B).   prec(B,A).   precneq(C,D).
precneq(E,C).   precneq(E,D).   precneq(F,A).   equiv(A∪E,A∪C).
```

Due to events' meaning, such order seems reasonable. If it is given as input to smodels, the answer set found includes the facts failsB and failsWC. This means that the given preference relation agrees with the basic axioms, however it cannot

be managed by using neither a Probability nor a Belief function. Nevertheless, one can use comparative Lower probabilities or comparative Plausibilities. ■

4 Partial-order extension

An interesting problem is that of finding an extension of a preference relation so as to take into account any further event extraneous "in some sense" to the initial assessment. Obviously, this should be achieved in a way that the extension retains the same character of the initial order (e.g., both satisfy the same axioms).

More precisely, let be given an initial (partial) assessment expressed as a set of known events $\mathcal{E}$ together with a (partial) order $\preceq$ over $\mathcal{E}$. Moreover, assume that $\preceq$ satisfies the axioms characterizing a specific class, say $\mathcal{C}$, of orders (cf. Sec. 3). Consider now a new event S (not in $\mathcal{E}$), implicitly described by means of a collection $\mathcal{C}'$ of set-theoretical constraints involving the known events. In the spirit of [8, Theorem 3], the problem we are going to tackle is: Determine which is the "minimal" extension $\preceq^+$ (over $\mathcal{E} \cup \{\mathsf{S}\}$) of the given preference relation $\preceq$, induced by the new event, which still belongs to the class $\mathcal{C}$. In other words, we are interested in ascertaining how the new event S must relate to the members of $\mathcal{E}$ in order that $\preceq^+$ still is in $\mathcal{C}$.

To this aim we want to determine the sub-collections $\mathcal{L}_\mathsf{S}, \mathcal{WL}_\mathsf{S}, \mathcal{U}_\mathsf{S}$, and $\mathcal{WU}_\mathsf{S}$, of $\mathcal{E}$ so defined:

$$
\begin{aligned}
E \in \mathcal{L}_\mathsf{S} &\text{ iff no extension } \preceq^* \text{ of } \preceq \text{ can infer that } \mathsf{S} \preceq^* E\\
E \in \mathcal{WL}_\mathsf{S} &\text{ iff no extension } \preceq^* \text{ of } \preceq \text{ can infer that } \mathsf{S} \prec^* E\\
E \in \mathcal{U}_\mathsf{S} &\text{ iff no extension } \preceq^* \text{ of } \preceq \text{ can infer that } E \preceq^* \mathsf{S}\\
E \in \mathcal{WU}_\mathsf{S} &\text{ iff no extension } \preceq^* \text{ of } \preceq \text{ can infer that } E \prec^* \mathsf{S}
\end{aligned}
$$

Consequently, any order $\preceq^+$ extending $\preceq$ must, at least, impose that:

$$
\begin{array}{ll}
E \prec^+ \mathsf{S} \text{ for each } E \in \mathcal{L}_\mathsf{S}, & E \preceq^+ \mathsf{S} \text{ for each } E \in \mathcal{WL}_\mathsf{S},\\
\mathsf{S} \prec^+ E \text{ for each } E \in \mathcal{U}_\mathsf{S}, \text{ and} & \mathsf{S} \preceq^+ E \text{ for each } E \in \mathcal{WU}_\mathsf{S},
\end{array}
$$

in order to satisfy the axioms characterizing $\mathcal{C}$.

In what follows, we describe an ASP-program that solves this problem by taking advantage from the computation executed during the classification phase (cf. Sec. 3): It gets as input the knowledge regarding the satisfied axiom(s), the preference and logical relations on the original set of events. Such program is fed to the solver, together with the description of the new event (see Example 3, below).

The handling of the axioms is done by ASP-rules of the form (here we list the rule for axiom (L), the other axioms are treated similarly):

```
:- holdsL, event(A), event(B), empty(N), empty(interset(A,B)),
   precneq(N,A), prec(unionset(A,B),B).
```

Rules of this kind (actually, constraints, in the sense described in Sec. 2), declare "undesirable" any extension for which the axiom is violated. For instance, consider a ground instance of the above rule; whenever the fact holdsL is present (i.e. is true in an answer set), then to make the (ground) clause satisfied, at

least one of the other literals must not belong to the answer set. (Notice that, these literals are all true exactly when (L) is violated.) Consequently, in order to activate this constraint (i.e. to impose axiom (L), for the case at hand) it suffices to add the fact holdsL to the input of the solver.

A further rule describes the potential answer set we are interested in:

```
1{ precneq(E1,E2), equiv(E1,E2), precneq(E2,E1) }1 :- event(E1), event(E2).
```

This rule simply asserts that any computed answer-set must predicate on each pair E1,E2 of events by stating exactly one, and only one, of the three facts precneq(E1,E2), equiv(E1,E2), and precneq(E2,E1). Then, smodels produces as output the answer sets fulfilling the desired requirements and encoding "legal" total orders.

The collections $\mathcal{L}_\mathsf{S}$, $\mathcal{WL}_\mathsf{S}$, $\mathcal{U}_\mathsf{S}$, and $\mathcal{WU}_\mathsf{S}$ can be obtained by computing the intersection *Cn* of all these answer sets. (Or, equivalently, by computing the set of logical consequences of the ASP-program. Notice that, in general, *Cn* needs not to be an answer set by itself.)

Unfortunately, not all the available ASP-solvers offer the direct computation of *Cn* as a built-in feature (DLV, for instance does, while smodels does not, cf. [1]). In general, a simple inspection of the answer sets generated by smodels allows one to detect which is the minimal extension of the preference relation which is mandatory for each total order.

In order to facilitate this detection, we designed a simple post-processor which filters smodels' output and produces the imposed extension of $\preceq$.

Example 3. Consider the partial order of Example 2 and the new event:

$$\mathsf{S} \equiv \textit{The real state of having SARS}$$

subject to these restrictions: $\mathsf{S}\subset\mathsf{F}$ and $\mathsf{F}\cap\mathsf{E}\subset\mathsf{S}$. Since in Example 2 we discovered that the initial preference relation satisfies axiom (PL), we want to impose such axiom and compute the extension of the initial order.

Once filtered smodels' output, we obtained the following result:[9]

precneq(S,A∪C)   precneq(S,A∪E)   precneq(S,D)
precneq(S,A)   precneq(S,Ω)   prec(∅,S)   prec(S,F)

showing that, apart from obvious relations induced by monotonicity, no significative constraint involving S can be inferred. Since S and E can be freely compared, this result suggests that either further investigation about relevance of the clinical test or a revision of the initial preference relation, should be performed. ■

The availability of automated tools able to extend preference orders, whenever new knowledge (new events) is acquired, directly suggests applications in expert systems and decision-support tools. In automated diagnosis, planning, or problem solving, to mention some examples, one could easily imagine scenarios where knowledge is not entirely available from the beginning. We could outline how a rudimental inference process could develop, by identifying the basic steps an automated agent should perform:

[9] We list here only the portion of the extension involving the new event S.

0) Acquisition of an initial collection of observations (events) about the object of the analysis, together with a (qualitative) partial preference assessment;
1) Detection of which is the most adequate (i.e., the most discriminant) uncertainty framework, through a "preference classification phase" (cf. Sec. 3);
2) Whenever new knowledge becomes available, refine agent's description of the real world by performing order extension (which substantially corresponds to knowledge inference. Cf. Sec. 4).

The results of step 2) could be then exploited to guide further investigations on the real world, in order to obtain new information. Then, step 2) will be repeated and the process will continue until further pieces of knowledge are obtainable or an enough accurate degree of believe is achieved.

Conclusions

In this paper we started an exploration of the potentialities offered by Answer Set Programming for building decision support systems based on qualitative judgments. Thanks to the remarkable features of ASP, the implementation of what could be thought as a kernel of an inference engine, sprouted almost naturally. Certainly, our research is at an initial stage and the implementation we reported on in this paper cannot be considered to be prototype. Next step in this research would consist in validating the proposed approach by means of a number of benchmarks aimed at testing our prototype on the ground of real applications. A comparison of its behaviour w.r.t. other possible declarative approaches, for instance exploiting Constraint Logic Programming, is due. Results of this activity will help in consolidating the prototype. In this context, a further goal consists in completing our approach so as to handle comparative Probabilities too. Since no axiomatic characterization of comparative Probabilities is known (up to our knowledge), this aim should be achieved through integration with efficient linear optimization tools (such as the column generation techniques). More in general, we envisage the design of a full-blown automated system which integrates different (in someway complementary) techniques and methods for uncertainty management; comprehending mixed numerical/qualitative assessments and conditional frameworks.

Acknowledgments

We thank the anonymous referees for the useful remarks, as well as Stefania Costantini and Agostino Dovier for fruitful discussions on the topics of this paper.

References

[1] Web references for some ASP solvers.
ASSAT: `http://assat.cs.ust.hk`
CCalc: `http://www.cs.utexas.edu/users/tag/cc`

Cmodels: http://www.cs.utexas.edu/users/tag/cmodels
DeReS: ftp://ftp.cs.engr.uky.edu/cs/software/logic
DLV: http://www.dbai.tuwien.ac.at/proj/dlv
Smodels: http://www.tcs.hut.fi/Software/smodels.

[2] C. Baral. *Knowledge representation, reasoning and declarative problem solving.* Cambridge University Press, 2003.

[3] T. Bilgiç. Fusing interval preferences. In *Proc. of EUROFUSE Workshop on Preference Modelling and Applications*, pages 253–258, 2001.

[4] A. Capotorti, G. Coletti, and B. Vantaggi. Non additive ordinal relations representable by lower or upper probabilities. *Kybernetika*, 34(1):79–90, 1998.

[5] A. Capotorti and B. Vantaggi. Relationships among ordinal relations on a finite set of events. In B. Bouchon-Meunier, R. R. Yager, and L. A. Zadeh, editors, *Information, Uncertainty, Fusion*, pages 447–458. Kluwer, 1998.

[6] A. Capotorti and B. Vantaggi. Axiomatic characterization of partial ordinal relations. *Internat. J. Approx. Reason.*, 24:207–219, 2000.

[7] G. Choquet. Theory of capacities. *Annales de l'institut Fourier*, 5:131–295, 1954.

[8] G. Coletti. Coherent qualitative probability. *J. Math. Psych.*, 34(3):297–310, 1990.

[9] E. Dantsin, T. Eiter, G. Gottlob, and A. Voronkov. Complexity and expressive power of logic programming. *ACM Computing Surveys*, 33(3):374–425, 2001.

[10] G. de Cooman. Confidence relations and ordinal information. *Inform. Sci.*, 104:241–278, 1997.

[11] G. de Cooman and P. Walley. The imprecise probability project. Available at :http://ippserv.rug.ac.be/.

[12] B. de Finetti. Sul significato soggettivo della probabilità. *Fundamenta Mathematicae*, 17:298–321, 1931. Engl. transl. in P. Monari and D. Cocchi, editors, *Induction and Probability*, CLUEB, Bologna: 291–321, 1993.

[13] B. de Finetti. *Theory of Probability.* Wiley, 1974. (Italian original *Teoria della probabilità*, Einaudi, Torino, 1970).

[14] D. Dubois. Belief structure, possibility theory and decomposable confidence measures on finite sets. *Comput. Artif. Intell.*, 5:403–416, 1986.

[15] D. Dubois, H. Fargier, and P. Perny. Qualitative decision theory with preference relations and comparative uncertainty: An axiomatic approach. *Artif. Intel.*, 148:219–260, 2003.

[16] D. Dubois, H. Fargier, and H. Prade. Decision-making under ordinal preferences and uncertainty. In *AAAI Spring Symposium on Qualitative Preferences in Deliberation and Practical Reasoning*, pages 41–46, 1997.

[17] D. Dubois and H. Prade. *Fuzzy sets and systems: theory and applications.* Academic Press, New York, 1980.

[18] D. Dubois, H. Prade, and R. Sabbadin. A possibilistic logic machinery for qualitative decision. In *AAAI Spring Symposium on Qualitative Preferences in Deliberation and Practical Reasoning*, pages 47–54, 1997.

[19] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *Proc. of 5th ILPS Conference*, pages 1070–1080, 1988.

[20] P. H. Giang and P. P. Shenoy. A comparison of axiomatic approaches to qualitative decision making under possibility theory. In *Proc. of the* 17th *Conf. on Uncertainty in Artificial Intelligence UAI01*, pages 162–170, 2001.

[21] G. J. Klir and T. A. Folger. *Fuzzy sets, uncertainty, and information.* Prentice-Hall, 1988.

[22] D. Lehmann. Generalized qualitative probability: Savage revisited. In *Proc. of the* 12th *Conf. on Uncertainty in Artificial Intelligence UAI96*, pages 381–388, 1996.

[23] V. Lifschitz. Answer set planning. In D. De Schreye, editor, *Proc. of ICLP'99*, pages 23–37. The MIT Press, 1999.
[24] W. Marek and M. Truszczyński. Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Springer, 1999.
[25] H. T. Nguyen and E. A. Walker. *A first course in fuzzy logic.* CRC Press, Boca Raton, 1997.
[26] G. Shafer. *A mathematical theory of evidence.* University of Princeton Press, Princeton, 1976.
[27] P. Simons. *Extending the Smodels System with Cardinality and Weight Constraints.* PhD thesis, Helsinki University, Department of Computer Science and Engineering, July 2000. Research report 58 of Technology Laboratory for Computer Science, HUT-TCS-A58.
[28] T. Syrjänen. Lparse 1.0 user's manual, 1999. Available at `http://www.tcs.hut.fi/Software/smodels`.
[29] P. Walley. *Statistical reasoning with imprecise probabilities.* Chapman and Hall, London, 1991.
[30] P. Walley. Measures of uncertainty in expert systems. *Artif. Intel.*, 83:1–58, 1996.
[31] P. Walley and T. Fine. Varieties of modal (classificatory) and comparative probability. *Synthese*, 41:321–374, 1979.
[32] S. K. M. Wong, Y. Y. Yao, P. Bollmann, and H. C. Bürger. Axiomatization of qualitative belief structure. *IEEE Trans. Systems Man Cybernet.*, 21:726–734, 1991.
[33] L. A. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

A gentle introduction to uncertainty measures

In this appendix we briefly describe the various generalizations of Probability measures used in this paper, as introduced in standard literature. The following material is far from being an exhaustive and complete treatment. We will give just a informal introduction to the subject. The interested reader can refer to the widely available literature. Introductory treatment of the relationships between Probability measures, belief functions and possibility measures, can be found in [21, 25, 30], to mention some among many.

We will consider, as domain of interest, a set Ω of possibilities (Ω is often refered to as *sample space*). For our purposes it is sufficient to consider the case of a finite domain. An *event* is then defined as a subset of Ω. In order to introduce uncertainty measures, we can consider any algebra $\mathcal{A}$ (on Ω), consisting of a set of subsets of Ω, such that $\Omega \in \mathcal{A}$, and closed under union and complementation.

All of the measures we are going to introduce will be (normalized) monotone real-valued functions over an algebra. Such functions are usually called (*Choquet*) *Capacities* [7], even if they are refered also as *fuzzy measures* or *Sugeno measures.*

Definition 1. *A real-valued function F on 2^{Ω} is a* Capacity *if it holds that*
$$F(\emptyset) = 0, \quad F(\Omega) = 1, \quad \text{and for all } A, B \subseteq \Omega \quad A \subseteq B \Longrightarrow F(A) \leqslant F(B).$$

Let us denote the class of Capacities over Ω by $CAP(\Omega)$.

The notion of Capacity is often too general to be of interest by itself. In fact adopting it, apart from monotonicity, there is no other relationship imposed between

the uncertainty assigned to a composed event, e.g. $F(A \cup B)$, and the uncertainty of its components $F(A)$ and $F(B)$. In order to reflect different rationales in managing the information, several constraints can be imposed on the manner in which uncertainties of composed events are determined. In what follows we describe some of the more interesting measures obtained by imposing further conditions on measures, apart to be a Capacity. We start with the most adopted measure of uncertainty. It is characterized by the additivity property of combination: A *Probability* P over Ω is a capacity which satisfies the following *additivity* requirement: For all $A, B \subseteq \Omega$ with $A \cap B = \emptyset$ $P(A \cup B) = P(A) + P(B)$. The class of all Probabilities over Ω is denoted by $PROB(\Omega)$. Clearly, we have that $PROB(\Omega) \subseteq CAP(\Omega)$. Let us introduce a further concept:

Definition 2. *Let F_1 and F_2 be two functions on 2^{Ω}. Then, F_1 is the* dual *of F_2 if for each $A \subseteq \Omega$ it holds that $F_1(A) = 1 - F_2(\Omega \setminus A)$.*

Note that the dual of a Capacity is a Capacity too. Moreover, the dual of a Probability is the Probability itself.

Additivity, even being widely adopted in "measurement" processes, is usually thought to be a too strong requirement. Hence, several generalizations have been proposed. In particular, the following definition characterizes those Capacities satisfying only one of the weak inequalities which, taken together, give additivity.

Definition 3. *Let Π and N be Capacities over Ω.*

- *Π is a* Possibility *measure (over Ω) if it satisfies the following property: For all $A, B \subseteq \Omega$ $\Pi(A \cup B) = \max\{\Pi(A), \Pi(B)\}$.*
- *N is a* Necessity *measure (over Ω) if it is the dual of a Possibility measure.*

It is immediate to see that

- a Possibility measure Π satisfies the *sub-additivity* property:
  For all $A, B \subseteq \Omega$ $\Pi(A \cup B) \leqslant \Pi(A) + \Pi(B)$;
- A Necessity measure N satisfies the *super-additivity* property:
  For all $A, B \subseteq \Omega$ $N(A \cup B) \geqslant N(A) + N(B)$.

A Possibility measure Π is usually induced by a *possibility distribution* (i.e. a fuzzy set) $\pi : \Omega \to [0, 1]$. The value $\pi(x)$ expresses the possibility of a singleton $x \in \Omega$ to be representative of the concept being considered. Possibility is then defined by putting $\Pi(A) = \max\{\pi(x) \mid x \in A\}$ for any $A \subseteq \Omega$. The classes of Possibilities and Necessities over Ω are denoted by $POS(\Omega)$ and $NEC(\Omega)$, respectively.

Let us consider now a slightly different situation. Suppose that the available (possibly incomplete) knowledge permits the formulation of some form of constraint on the Probability of the events. Ideally, such constraints may determine a unique Probability measure. In general, this is not the case. In fact, there may be a non-void set of Probability measures which satisfy the given constraints. Here we describe the measures induced by such set. In particular, a set of Probabilities measures (over Ω) induces two natural measures. Namely, its lower and upper envelope.

Definition 4. *Let $\emptyset \neq \mathcal{P} \subseteq PROB(\Omega)$. The* lower envelop $\underline{\mathcal{P}}$ *and the* upper envelopes $\overline{\mathcal{P}}$ *of $\mathcal{P}$ are defined as:*

- *For each $A \subseteq \Omega$, $\underline{\mathcal{P}}(A) = inf\{P(A) \mid P \in \mathcal{P}\}$;*
- *For each $A \subseteq \Omega$, $\overline{\mathcal{P}}(A) = sup\{P(A) \mid P \in \mathcal{P}\}$.*

Lower envelopes are usually called Lower probability measures*, while upper envelopes, which are their duals, are called* Upper probability measures*.*

Let us denote the classes of Lower and Upper probabilities over Ω by $LOWP(\Omega)$ and $UPP(\Omega)$, respectively.

It remains to introduce Belief and Plausibility measures. With the most general formulation, following [26], we have:

Definition 5. *A function* $Bel : 2^{\Omega} \rightarrow [0,1]$ *is a* Belief *measure if it is a Capacity and it satisfies the following condition (known as* ∞-monotonicity*).*

For each $n \geqslant 1$, $Bel\big(\bigcup_{i=1}^{n} A_i\big) \geqslant \sum_{\emptyset \neq I \subseteq \{1,\ldots,n\}} (-1)^{|I|+1} Bel\big(\bigcap_{i\in I} A_i\big)$

(where $A_i \subseteq \Omega$ *for each* i*).*

Intuitively speaking, a Belief function Bel is usually constructed through a basic assignment of uncertainty, not necessarily being a Capacity, $\mu : 2^{\Omega} \rightarrow [0,1]$ so that, for any proposition $A \subseteq \Omega$, $Bel(A) = \sum_{B \subseteq A} \mu(B)$.

Notice that Belief functions are often called *Capacities monotone of infinite order*. Capacities which satisfy the above condition with the restriction that $n \leqslant N$ are then said *monotone of order* N (or N-monotone). Dually, if the opposite inequality ($\leqslant$) is considered, the measure is said to be an N*-alternating* capacity. For $N = 2$ these properties reduce to usual super- and sub-additivity, respectively.

The dual of a Belief measure is called *Plausibility* measure. The classes of Belief measures and Plausibility measures are denoted by $BEL(\Omega)$ and $PL(\Omega)$, respectively.

The following relationships can be shown to hold between the classes of Capacities seen so far:

$$CAP(\Omega) \supset LOWP(\Omega) \supset BEL(\Omega) \supset NEC(\Omega)$$
$$CAP(\Omega) \supset UPP(\Omega) \supset PL(\Omega) \supset POS(\Omega)$$
$$BEL(\Omega) \cap PL(\Omega) \supset PROB(\Omega).$$

SAT-based Analysis of Cellular Automata

Massimo D'Antonio and Giorgio Delzanno

Dip. di Informatica e Scienze dell'Informazione - Università di Genova via Dodecaneso 35, 16146 Genova, Italy, e-mail: giorgio@disi.unige.it

Abstract. Cellular Automata are a powerful formal model for describing physical and computational processes. Qualitative analysis of Cellular Automata is in general a hard problem. In this paper we will investigate the applicability of modern SAT solvers to this problem. For this purpose we will define an encoding of reachability problems for Cellular Automata into SAT. The encoding is built in a modular way and can be used to test *inverse reachability* problems in a natural way. In the paper we will present experimental results obtained using the SAT-solver zChaff.

1 Introduction

Cellular Automata (CAs) [18] are decentralized spatial extended systems consisting of large numbers of simple identical components with local connectivity. Such systems have the potential to perform complex computations with a high degree of efficiency and robustness, as well as to model the behavior of complex systems in nature. For these reasons CAs have been studied extensively in natural sciences, mathematics, and in computer science. For instance, they have been used as parallel computing devices for image processing, and as abstract models for studying cooperative or collective behavior in complex systems (see e.g. [2,3,21,11]).

Quantitative vs Qualitative Analysis Several tools have been used to animate specifications of CAs and to perform statistical analysis of their behavior, e.g., for car traffic or artificial life processes simulation (for a survey see e.g. [22]). Simulation amounts to compute all possible reachable configurations. For deterministic CAs this operation is fairly simple but some care must be taken in the way the transition rule is stored.

Differently from plain simulation, a *qualitative analysis* of the behavior of a CA can be a difficult problem to solve. In fact, problems like reachability of subconfigurations or existence of a predecessor configuration for generic CAs have been shown to be exponentially hard [6,19,20]. The hardness of some computational problems for CAs has been exploited for interesting applications. As an example, reversibility (undecidable in general [10]) has been used for designing cryptographic systems based on CAs [9].

Towards a Practical Solution of Hard Problems In recent years practical solutions to large instances of known hard problems like SAT (satisfiability of propositional formulas, a well known NP-complete problem) have been made possible by the application of specialized search algorithms and pruning heuristics. The connection between the complexity of some interesting problems for CAs and SAT (see e.g. [6]) suggests us a possible new application of all these technologies.

Technical Contribution In this paper we will investigate in fact the applicability of SAT solvers to the *qualitative analysis* of CAs. Specifically, following the *Bounded Model Checking* (BMC) philosophy introduced in [1], we will define a *polynomial time* encoding of a *bounded number* of evolution steps of a CA into a formula in *propositional logic*. Our encoding allows us to specify in a *declarative* and *modular way* several decision problems for CAs like (inverse) reachability as a SAT problem. As a result, we obtain an effective verification procedure for the analysis of CAs by resorting to efficient-in-practice existing SAT solvers like [7,8,16,14].

As preliminary experiments, we have selected a non-trivial example of CA, namely Mazoyer's solution to the Firing Squad Synchronization Problem [12]. In this example the length of the evolution (the diameter of the model in the terminology of BMC) leading to a successful final configuration depends on the dimension of the cellular space. For this reason, this problem is adequate to check qualitative problems that can be encoded as *bounded reachability*, e.g., checking if the final solution is the correct one, computing predecessor configurations, computing alternative initial configurations leading to the same solution, etc. We will use this case-study to test the performance of one of the fastest existing SAT-solvers called zChaff [14]. zChaff [14] can handle problems with up to 10^6 propositional variables. In our setting zChaff returns interesting results for problems of reasonable size (e.g. cellular spaces with 70 cells and evolution of 140 steps). Some built-in heuristics of zChaff however turned out to be inadequate for CA-problems like inverse reachability. We believe that specialization of SAT-solving algorithms to problems formulated on CAs could be an interesting future direction of research.

Plan of the Paper In Section 2 we introduce the notion of Cellular Automata. In Section 3 we define the encoding of the evolution of a CA and of its qualitative problems into a SAT formula. In Section 4 we encode several interesting properties of CAs. In Section 5 we discuss experimental results obtained with existing solvers. In Section 6 we discuss related works and future directions of this research.

2 Cellular Automata (CAs)

A CA consists of two components. The first component is a *cellular space*, i.e. a collection of identical finite-state machines (cells), each with an identical pattern

of local connection to other cells. Let S be the set of states of each automata. Each cell is denoted by an index $\boldsymbol{i}$. The *neighborhood* of a cell i is the set of cells of the network which will locally determine the evolution of i.

The second component is a transition rule δ that gives the update state for each cell $\boldsymbol{i}$ in function of the state of the cells in its neighborhood. In a CA a *discrete global clock* provides an update signal for all cells: at each time step all cells update their states synchronously according to δ.

Formally, we have the following definition.

Definition 1. *A d-CA $\mathcal{A}$ is a tuple $\langle d, S, N, \delta \rangle$ where*

- *$d \in \mathbb{N}$ is the dimension of the cellular space ($\mathbb{Z}^d$);*
- *S is the finite set of states of $\mathcal{A}$;*
- *$N \subseteq \mathbb{Z}^d$ is the* neighborhood *of $\mathcal{A}$ and $\mid N \mid = n$;*
- *$\delta : \ S^{n+1} \to S$ is the transition rule of $\mathcal{A}$.*

The meaning of the neighborhood of $\mathcal{A}$ is as follows. Suppose that $N = \{\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}\}$. Given a cell $\boldsymbol{i}$, its neighborhood is obtained then by considering the cells $\boldsymbol{i}, \boldsymbol{i} + \boldsymbol{v_1}, \ldots, \boldsymbol{i} + \boldsymbol{v_n}\}$. A classical examples is von Neumann's neighborhood defined as $N_N = \{(1,0), (0,1), (-1,0), (0,-1)\}$ for 2-CA.

2.1 Evolution of a CA

The computational meaning of a d-CA $\mathcal{A} = \langle d, S, N, \delta \rangle$ with $N = \{\boldsymbol{v_1}, \ldots, \boldsymbol{v_n}\}$ is defined as follows.

A *configuration* of a $\mathcal{A}$ is defined as a function $c : \mathbb{Z}^d \to S$ that assigns a state to each cell of the cellular space. We will use $\mathcal{C}$ to denote the set of configurations.

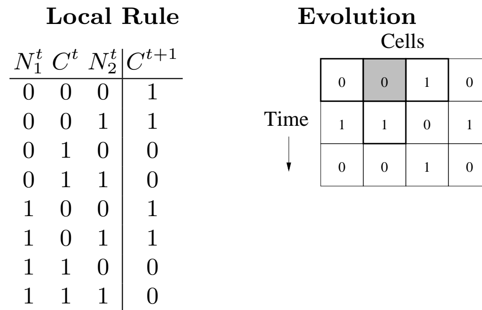
The *global evolution function* $G_{\mathcal{A}}$ associated to $\mathcal{A}$ is a transformation from configurations to configurations such that

$$G_{\mathcal{A}}(c)(\boldsymbol{i}) = \delta(\langle c(\boldsymbol{i}), c(\boldsymbol{i} + \boldsymbol{v}_1), ..., c(\boldsymbol{i} + \boldsymbol{v}_n)\rangle) \textit{ for any } c \in \mathcal{C}, \ \boldsymbol{i} \in \mathbb{Z}^d.$$

Given an initial configuration c_0, the evolution of $\mathcal{A}$, written $Ev^{\mathcal{A}}(c_0)$, is a sequence $\{c_t\}_{t \geq 0}$ of configurations such that $c_{t+1} = G_{\mathcal{A}}(c_t)$ for any $t \geq 0$. A configuration c' is *reachable* in k steps from configuration c_0 if there exists an evolution $\{c_t\}_{t \geq 0}$ such that $c' = c_k$. A configuration c' is *reachable* from c_0 if it is reachable in $k \geq 0$ steps.

Previous definitions are for infinite cellular spaces but computer simulations are obviously constrained on finite spaces, hence periodic boundary conditions (for example, the cellular space is defined as a ring) or the definition of constant boundary cells are necessary.

Example 1. Fig. 1 illustrates an example of local rule defined over $S = \{0,1\}$ and $N = \{-1,1\}$, given in a tabular form, and of an evolution on a circular cellular space. In the table representing the local rule the two columns C^t and C^{t+1} denote the state of a cell at time t and $t+1$ respectively, whereas columns N_1^t and N_2^t denote the state of cells in the neighborhood. In the evolution we highlight the neighborhood to which a rule is applied.



Local Rule

N_1^t	C^t	N_2^t	C^{t+1}
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

Fig. 1. A simple CA.

2.2 Expressiveness and Computational Complexity

CAs are a powerful computational model. In fact, it is sufficient to consider one-dimensional CAs to simulate Turing Machines. Given the richness of the model, several decision problems related to the computational interpretation of the evolution of CAs are hard or impossible to solve. The hardest problems are often related to the inverse exploration of the configuration space of a CA. Note, in fact, that the *past history* of a configuration contains non-deterministic choices for its predecessors. For instance, let us consider the Predecessor Existence Problem, defined as follows:

(PEP) Given a d-CA $\mathcal{A}$ *and* $x \in \mathcal{C}$, $\exists y \in \mathcal{C}$ *such that* $x = G_{\mathcal{A}}(y)$*?*

PEP is in NP for “finite” d-CAs, NP-complete for $d > 1$, and NLOG for 1-CAs. Proofs of NP-completeness of this kind of problems are often given via reductions of 3SAT into the decision problem taken into consideration. In this paper we will try to exploit reductions going in the opposite direction, namely from CA to SAT, in order to obtain an effective and flexible method for the analysis of difficult decision problems for CAs.

3 From Cellular Automata to Propositional Logic

In this section we will define a representation of a finite prefix of the evolution of a CA in propositional logic. The resulting formula will be used later to define several interesting decision problems for CAs in a formal and modular way. Furthermore, it represents the first step towards the use of SAT solvers for the analysis of CAs.

Symbolic Representation of Configurations Given a CA $\mathcal{A}$, we first represent its set of states $S = \{s_1, \ldots, s_n\}$ using a binary encoding over $m = \lceil log_2 n \rceil$ bits of a given choice of ordering numbers. Let us call $S' = \{w_1, \ldots, w_n\}$ be the resulting set of binary representations, where each $w_i \in \{0,1\}^m$ (sequence

of m bits). As an example, for $S = \{red, blue, yellow\}$ we can use 2 bits and the following encoding $w_{red} = 00$, $w_{blue} = 01$, and $w_{yellow} = 10$. Every state represented in binary can be naturally encoded as a propositional formula as follows. Let ℓ be a label (i.e. a string used later for stamping predicate symbols with time and position) and w a sequence of m bits $b_1 \dots b_m$. Furthermore, let . be an operator for concatenating labels. Then, we define the formula encoding b as follows:

$$Cod_w(w, \ell) \doteq \bigwedge_{i=1}^{m} Cod_b(b_i, i.\ell) \text{ where } Cod_b(b, \ell) \doteq \begin{cases} x_\ell \ if \ b = 1 \\ \neg x_\ell \ \ if \ b = 0 \end{cases}$$

Going back to our example, assume that 0.0 represents a cell in position 0 at time 0, then $Cod_w(w_{blue}, 0.0) = \neg x_{1.0.0} \wedge x_{2.0.0}$. Generalizing this idea, in the following we will use strings of the form $p.t$, where p is denotes a position and t a time-stamp, as labels for encoding the evolution of configurations using propositional formulas.

Specifically, let us assume that the cellular space is linearized into the range $1, \dots, N$. Then, a configuration is simply a tuple $c = \langle\langle 1, w_1\rangle, \dots, \langle N, w_N\rangle\rangle$, where w_i is the binary representation of a state of $\mathcal{A}$.

The encoding of c at instant t is defined then as follows:

$$Cod_c(c, t) \doteq Cod_w(w_1, 1.t) \wedge \dots \wedge Cod_w(w_N, N.t)$$

Thus, $Cod_c(c, t)$ gives rise to a conjunction of *literals* over the set of predicate symbols $x_{b.p.t}$ where $b \in \{1, \dots, m\}$, $p \in \{1, \dots, N\}$. For instance, the encoding of the configuration of $c_0 = \langle\langle 1, w_{red}\rangle, \langle 2, w_{red}\rangle, \langle 3, w_{blue}\rangle\rangle$ is the formula

$$\underbrace{\neg x_{1.1.0} \wedge \neg x_{2.1.0}}_{red} \wedge \underbrace{\neg x_{1.2.0} \wedge \neg x_{2.2.0}}_{red} \wedge \underbrace{\neg x_{1.3.0} \wedge x_{2.3.0}}_{blue}$$

We are ready now to encode a transition rule.

Symbolic Representation of the Transition Rules A CA is usually given in form of a table: each row is transition rule for one possible global state of the neighborhood of a generic cell. Let us call $\mathcal{R}_p$ the set of rows of the table relative to a cell in position p. Let us assume that the neighborhood is $v_1, \dots, v_n$. Let $R \in \mathcal{R}_p$ be a rule that, at time t, operates on the neighborhood of a cell p, namely $\langle\langle p, w\rangle, \langle(p + v_1, w_1\rangle, \dots, \langle p + v_n, w_n\rangle\rangle$, and that updates its state into w'. Then, the encoding of R is the formula

$$Cod_r(R, p, t) \doteq Cod_w(w, p.t) \wedge \bigwedge_{i=1}^{n} Cod_w(w_i, (p + v_i).t) \wedge Cod_w(w', p.(t + 1))$$

We can extend this encoding to $\mathcal{R}_p$ in the natural way:

$$Cod_R(\mathcal{R}_p, t) \doteq \bigvee_{R \in \mathcal{R}_p} Cod_r(R, p, t)$$

Using this disjunctive formula we can express one evolution step without having to specify the initial configuration (we let open all possible choices of rules in $\mathcal{R}_p$). Note that to obtain a total transition function with respect to the set of variables used in the encoding we need to add identity rules

Symbolic Representation of the CA-Evolution Specifically, the formula $Ev^{\mathcal{A}}(N,k)$ that describes all possible evolutions in k steps of a CA with N cells is defined as follows

$$Ev^{\mathcal{A}}(N,k) \doteq \bigwedge_{t=0}^{k-1} \bigwedge_{i=1}^{N} Cod_R(\mathcal{R}_i, t)$$

Note that the formula $Ev^{\mathcal{A}}(N,k)$ is not in conjunctive normal form (CNF). The following properties formalize the connection between the evolution of a CA and the formula $Ev^{\mathcal{A}}(N,k)$.

Proposition 1. *Given a CA $\mathcal{A}$, every assignment ρ satisfying of the formula $Ev^{\mathcal{A}}(N,k)$ represents a possible evolution $\{c_t\}_{t\geq 0}$ of $\mathcal{A}$ such that ρ satisfies $Cod(c_t,t)$ for any $t \geq 0$.*

As a consequence, we have the following link between k-reachability and satisfiability of $Ev^{\mathcal{A}}(N,k)$.

Theorem 1. *Given a CA $\mathcal{A}$ and two configurations c and c', c' is reachable in k-steps from c if and only if the formula*

$$REACH_k \doteq Cod_c(c,0) \wedge Ev^{\mathcal{A}}(N,k) \wedge Cod_c(c',k)$$

is satisfiable.

As a final remark, it is easy to check that the size of the encoding is polynomial in the size of the cellular space, size of the neighborhood and in the number of steps taken into consideration.

Example 2. Let $\mathcal{A}$ be a 1-CA, with $S = \{0,1\}$, circular boundary conditions and neighborhood $I = \{-1\}$ (i.e. we only look at the left neighbor's cell) and rule described in Fig. 2. In the left table of Fig. 2 we use the variable x^{it} to denote the value of cell i a time t. The column x^{it+1} denotes the new state of cell i at time $t+1$. The right table of Fig. 2 shows the corresponding encoding of the local rule when interpreting 0 and 1 as truth values. A configuration $c = \langle 0,1\rangle$ with 2 cells at time 0 can be encoded as the conjunction of literals $\neg x_{1.0} \wedge x_{2.0}$. Thus, the evolution from $t=0$ to $t=1$ of cell c can be represented as follows

$$\neg x_{1.0} \wedge x_{2.0} \wedge \begin{pmatrix} \neg x_{1.0} \wedge \neg x_{2.0} \wedge x_{2.1} \\ \vee \\ \neg x_{1.0} \wedge x_{2.0} \wedge \neg x_{2.1} \\ \vee \\ x_{1.0} \wedge \neg x_{2.0} \wedge \neg x_{2.1} \\ \vee \\ x_{1.0} \wedge x_{2.0} \wedge x_{2.1} \end{pmatrix} \wedge \begin{pmatrix} \neg x_{2.0} \wedge \neg x_{1.0} \wedge x_{1.1} \\ \vee \\ \neg x_{2.0} \wedge x_{1.0} \wedge \neg x_{1.1} \\ \vee \\ x_{2.0} \wedge \neg x_{1.0} \wedge \neg x_{1.1} \\ \vee \\ x_{2.0} \wedge x_{1.0} \wedge x_{1.1} \end{pmatrix} \wedge \neg x_{1.1} \wedge \neg x_{2.1}$$

x^{i-1t}	x^{it}	x^{it+1}
0	0	1
0	1	0
1	0	0
1	1	1

$$\Rightarrow \begin{array}{c} (\neg x_{i-1.t} \wedge \neg x_{i.t} \wedge x_{i.t+1}) \\ \vee \\ (\neg x_{i-1.t} \wedge x_{i.t} \wedge \neg x_{i.t+1}) \\ \vee \\ (x_{i-1.t} \wedge \neg x_{i.t} \wedge \neg x_{i.t+1}) \\ \vee \\ (x_{i-1.t} \wedge x_{i.t} \wedge x_{i.t+1}) \end{array}$$

Fig. 2. Encoding of the local rule.

As expected, the valuation

$$v : \{x_{1.0} \mapsto F, x_{2.0} \mapsto T, x_{1.1} \mapsto F, x_{2.1} \mapsto F\}$$

satisfies the resulting propositional formula. If we do not specify the final configuration, a solution to the SAT-problem

$$\neg x_{1.0} \wedge x_{2.0} \wedge \begin{pmatrix} \neg x_{1.0} \wedge \neg x_{2.0} \wedge x_{2.1} \\ \vee \\ \neg x_{1.0} \wedge x_{2.0} \wedge \neg x_{2.1} \\ \vee \\ x_{1.0} \wedge \neg x_{2.0} \wedge \neg x_{2.1} \\ \vee \\ x_{1.0} \wedge x_{2.0} \wedge x_{2.1} \end{pmatrix} \wedge \begin{pmatrix} \neg x_{2.0} \wedge \neg x_{10} \wedge x_{1.1} \\ \vee \\ \neg x_{2.0} \wedge x_{1.0} \wedge \neg x_{1.1} \\ \vee \\ x_{2.0} \wedge \neg x_{1.0} \wedge \neg x_{1.1} \\ \vee \\ x_{2.0} \wedge x_{1.0} \wedge x_{1.1} \end{pmatrix}$$

allows us to compute the successor configuration by simply projecting the resulting valuation on variables stamped with time index 1. In the following section we will formalize more precisely how to specify reachability properties using SAT-encoding of CA-evolutions.

4 SAT-based Qualitative Reasoning

The formula $Ev^{\mathcal{A}}(N, k)$ represents an encoding of all possible CA-evolutions of length k independently from any initial or target configuration. This property allows us to encode in a modular way several interesting properties of CAs in terms of satisfiability of a propositional formula. In the rest of the section we will discuss some examples.

Reachability Given an initial configuration c and a configuration c', we can decide whether c' is reachable in k-steps from c by solving the satisfiability problem for the formula

$$REACH_k \doteq Cod_c(c, 0) \wedge Ev^{\mathcal{A}}(N, k) \wedge Cod_c(c', k)$$

Actually, we can also compute all configurations reachable in at most k-steps. We first solve the satisfiability problem for the formula

$$CREACH_k \doteq Cod_c(c, 0) \wedge Ev^{\mathcal{A}}(N, k)$$

and then extract the configurations c_t for $0 \leq t \leq k$ from the resulting satisfying assignment ρ.

Note that the formula $CREACH_k$ is always satisfiable if the rule table contains a totally defined rule, otherwise we have to choose accurately the initial state.
$(C)REACH_k$ can be refined in order to be satisfiable only if the evolution is acyclic. The acyclicity test $ACYCLIC_k$ amounts to require that the assignment to predicates at time t is distinct from all assignments at time $t' < t$ for any pair of values of t and t' between 0 and k. Thus, for finite CA we can explore all the reachable configurations by *iterative deepening* on k until the $ACYCLIC_k$ fails. This way we can solve reachability problems and compute the reachability set of a CA.

Inverse Reachability Compared to approaches based on simulation, a distinguishing feature of our encoding is that the formula $Ev^{\mathcal{A}}(N,k)$ is independent from the search strategy (e.g. forward exploration/backward exploration). we adopt for the analysis of a CA. This feature makes easy the encoding of *inverse* reachability problems in terms of reachability. For instance, given a (sub)configuration c and a configuration c', we can decide whether c' is a predecessor of c by solving the satisfiability problem for the formula

$$PEP \doteq Cod_c(c',0) \wedge Ev^{\mathcal{A}}(N,1) \wedge Cod_c(c,1)$$

Note that, if c is a subconfiguration, in its encoding all unspecified cells will be represented with predicates without constraints on its truth values. Similarly, given a (sub)configuration c and a configuration c', we can decide whether c' is a predecessor in k-steps of c by solving the satisfiability problem for the formula

$$PREP_k \doteq Cod_c(c',0) \wedge Ev^{\mathcal{A}}(N,k) \wedge Cod_c(c,k)$$

Finally, as for forward reachability, we can also *compute* a possible trace in the CA-evolution (and the corresponding initial state) of k steps that leads to an encoded configuration c at time k. We first solve the satisfiability problem for the formula

$$IREACH_k \doteq Ev^{\mathcal{A}}(N,k) \wedge Cod_c(c,k)$$

and then extract the configurations c_t for $0 \leq t \leq k$ from the resulting satisfying assignment ρ. In order to find the set of all predecessors of a configuration c we can use the following procedure: (1) set F to $IREACH_1$; (2) solve the satisfiability problem for the formula F (that gives us as a result *one* possible predecessor); (3) if the problem is unsatisfiable exit the procedure, otherwise (4) extract the formula G corresponding to the computed predecessor, set F to $F \wedge \neg G$ and go back to (2).

4.1 Goal-driven SAT-encoding

Inverse reachability is the more difficult problem among the one listed in the previous section. This is due to the non-determinism in the computation of the

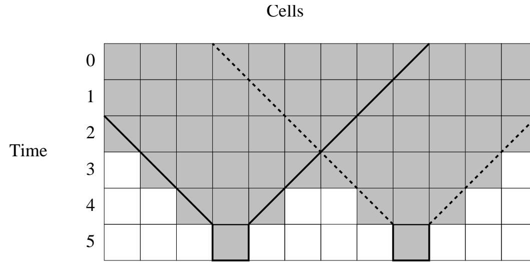


Fig. 3. Example of cone of influence associated to a given final subconfiguration.

$$cone(i, 0) = \{\langle i, 0\rangle\}$$
$$cone(i, t) = \{\langle i, t\rangle\} \cup cone(i, t-1) \cup \bigcup_{j=1}^{n} cone(i + v_j, t-1), \text{ for } t > 0$$

Fig. 4. Definition of the cone of influence of a given cell.

preimage of a given configuration. To reduce the complexity of the SAT-solving procedure we can try to reduce the size of the SAT-formula encoding the CA-evolution and specialize it to the goal we are looking for. For example, suppose that for a 1-CA with N cells we want to compute the initial (sub)configurations that lead to a certain state for cell i in k steps. To optimize the encoding of this problem, we can apply a version of the *cone of influence* introduced in [1] to statically compute the set of variables that influence the evolution of cell i. In Fig. 3 an example of *cone of influence* is presented for 2 cells after 5 evolution-step of a 1-CA with neighborhood of unitary radius. The gray zone shows the union of the useful cones for the considered cells. The Fig. 4 we define a procedure *cone* for computing the cone of influence associated to a given cell i at time t. The procedure only depends on the definition of the neighborhood. Our goal is to reduce the number of variables in the encoding of the CA-evolution by choosing only necessary cells for the encoded properties. Suppose that c is the subconfiguration for which we want to compute the predecessors in k-steps. Then, the set of variables computed by the algorithm in Fig. 4 can be used during the construction of the SAT-formula as follows. We first construct the cone of influence for all cells i contained in c. Then, we generate the formula $\mathbf{Cod_r}(\mathcal{R}, i', t')$ if and only if $\langle i', t'+1\rangle \in cone(i, t)$. The quality of this heuristic clearly depends on the *locality* of the neighborhood and on the final subconfiguration.

To limit the number of variables in the SAT-formula, we can exploit the fact that *boundary* cells (i.e. cells that encode the boundary of the cellular space) never change state. Thus, we only need to encode boundary cells at time zero and refer to this encoding in every step of the construction of the SAT-formula. This optimization preserves the correctness of the encoding.

In the following section we will discuss a practical evaluation of the proposed SAT-based methodology and related heuristics/optimizations.

5 Experimental Results

In order to test the effectiveness of the SAT-based analysis we have performed several experiments using the SAT-solver zChaff [14]. zChaff implements the Davis-Putnam algorithm [5] and it is considered as one of the fastest existing solvers. In general zChaff manages formulas with about 10^6 propositional variables and about 10^7 clauses. In some preliminary experiments presented in [4] we have compared zChaff with other solvers like ICS [8], SIMO [16] and HeerHugo [7] on problems related to CAs. zChaff always gave us the best results in terms of execution time. The input for zChaff is a CNF-formula written in DIMACS format. By using the structure-preserving algorithm of [15], we have built a front end to put the formula resulting from the encoding of a CA-evolution in CNF. The algorithm makes use of a polynomial number of auxiliary variables (one per each row of a CA-table). All experiments are performed on a Pentium4 2 GHz, with 1Gb of RAM.

5.1 Tested Example

As main example we have considered a solution to the Firing Squad Synchronization Problem (FSSP). FSSP was introduced by Moore in [13]. One considers here a finite ordered line of n finite-state machines. At time 0, the leftmost cell is distinguished (general) from the others (soldiers). These machines work synchronously; the state of a machine i at time $t+1$ depends only on the states at time t of the machines $i-1$, i and $i+1$. The problem is to define finite sets of states and transition rules so that all machines enter for the first time a distinguished state (fire) at the very same moment. This problem can be solved by defining a 1-CA with n cells representing the firing squad and the general. Mazoyer [12] has given a six-state (plus a cell for the boundary of the cellular space) minimal time solution in which the general creates two *waves* that propagates through the squad at different speed so as to reach a solution in exactly $2 * \#cells - 2$. This problem is thus adequate for testing *bounded reachability* problems. Mazoyer's CA is defined via 120 interesting rows (the total transition relation has 7^3 rows, the remaining 223 rows do not change the cell state).

5.2 Tested Properties

In Table 1 we illustrate the type of reachability properties we have tested on FSSP. Specifically, we have considered reachability problems in which either the initial and final configuration are completely specified or part of them are left unconstrained. For instance, $I^{-n}F$ denotes a reachability problem in which n cells of the initial configuration are left unconstrained (i.e. we considered a subconfiguration of the initial configuration); F denotes a problem in which only the final state is specified. The properties $nF+B$ and $F+B+nL$ listed in Table 1 are related to special tricks we used to exploit an heuristic called VSIDS of zChaff. The heuristic VSIDS is used to choose a starting variable for the resolution algorithm between those that appear most frequently in the

Added to $Ev^{\mathcal{A}}$	Description
I	Initial configuration (i.e. *CREACH*).
I^{-n}	Initial subconfiguration in which n cells are unconstrained.
F	Final configuration (i.e. *IREACH*)
F^1	One cell of the final configuration.
B	Boundary cells.
$I + F$	Initial configuration (i.e. *REACH*).
$F + B$	Final configuration and boundary cells.
$I^{-n} + F$	Initial configuration without n cells and a final configuration.
$F^1 + B$	Only one cell of the final configuration and boundary cells
$nF + B$	Final configuration with n-copies of formula F.
$F + B + nL$	$F + B$ and n-copies of the last step of the evolution formula.

Table 1. List of reachability properties considered in the experiments.

formula. In order to exploit this heuristic we can either put n-copies of the formula encoding F (property $nF + B$) or put n-copies of the last step of the formula representing the evolution (i.e. the clauses leading to variables occurring in F) (property $F + B + nL$). This way when computing predecessors of a configuration we force the SAT-solver to choose the variables that encode the final configuration, i.e., those with the smaller number of occurrences in $Ev^{\mathcal{A}}$ but with trivial truth assignment.

5.3 Outcomes of the Experiments

All the experiments require a preliminary compilation phase in which the SAT-formula is built up starting from a CA rule table. The time required for the biggest example is around 20 minutes due to the huge size of the resulting output. In the following we will focus however on the performance of the solver on SAT-formula of different size and on properties taken from Table 1.

$I + F$ and $I^{-n} + F$ Properties In a first series of experiments we have tested $I + F$-like properties on the CA-solution to FSSP. The results are shown in Table 2. The final configuration considered here is the one in which all soldiers in the firing squad have received the fire command. For this kind of problems, the size of the formulas that zChaff manages to solve scales up smoothly to formulas with one million variables. For instance, on a cellular space of dimension 70 and with 138 evolution steps it requires 1 minute to check that the CA solves FSSP. We considered then problems of the form $I^{-n}F$. Since there might be several initial states (legal or illegal) containing the subconfiguration I^{-n} and leading to the same final state F, the resulting SAT problem becomes more difficult. As expected, on this new kind of problems the performance of zChaff decreases with the number of cells n removed from I. As an example, for a CA with 15 cells (hence 28 steps) it takes more $11m$ to solve the $I^{-4} + F$ problem.

Problem	#Cells	#Steps	Input(MB)	#Vars	#Clauses	ExTime
$I+F$	15	28	12.18	51708	655713	3s
$I+F$	29	56	51.76	199842	2535241	13s
$I+F$	40	78	101.8	383883	4870563	25s
$I+F$	50	98	165.57	602853	7649203	39s
$I+F$	70	138	340.21	1118393	15079683	1m 22s
$I^{-1}+F$	15	28	12.18	51708	655710	3s
$I^{-2}+F$	15	28	12.18	51708	655707	69s
$I^{-3}+F$	15	28	12.18	51708	655704	65s
$I^{-4}+F$	15	28	12.18	51708	655701	30m53s

Table 2. Experiments on $I + F$-like problems.

Problem	#Cells	#Steps	Input(MB)	#Vars	#Clauses	ExTime
I	15	28	12.18	51708	655668	3s
I	29	56	51.76	199842	2535154	13s
I	50	98	165.57	602853	7649053	40s
I	70	138	340.21	1118393	15079473	1m 06s
I^{-1}	15	28	12.18	51708	655665	3s
I^{-2}	15	28	12.18	51708	655662	7m 21s
I^{-3}	15	28	12.18	51708	655659	4s
I^{-4}	15	28	12.18	51708	655656	10s
I^{-5}	15	28	12.18	51708	655653	3m 22s
I^{-7}	15	28	12.18	51708	655647	35m 36s

Table 3. Experiments on I and I^{-n} problems.

I and I^{-n} Properties In a second series of experiments we have tested I-like properties that can be used to *compute* reachable states. The results are shown in Table 3. Unexpectedly, the behavior of zChaff is quite irregular with respect to the growth of the size of the SAT formulas. The average of the execution times tends to grow exponentially with the number of cells removed from the initial configuration until the problem becomes trivial (i.e. when we do not have neither I nor F). Other examples we tested in [4] did not suffer from this anomaly.

F Properties In the third series of experiments we have considered different types of F-like properties that can be used to compute *predecessor configurations* of a given final configuration. This is a hard problem in general. As expected, we had to reduce the size of the SAT-formulas in order to get reasonable execution times. In Table 4 we have considered inverse reachability starting from F. For a cellular space of dimension 10 it takes about $9m$ to solve the problem F. Adding a constraint on the boundary cells, i.e. property $F + B$, we dramatically decrease the execution time ($2m$). In this example zChaff infers the correct initial

Problem	#Cells	#Steps	Input(MB)	#Vars	#Clauses	ExTime
F	10	18	5	22173	281010	9m 40s
$F+B$	10	18	5	22173	281013	2m 27s
$F+B+2L$	10	18	5	22173	296623	35s
$10F+B$	10	18	5	22173	281043	39s
$400F+B$	10	18	5	22173	292983	1m 50s

Table 4. Experiments on F-like problems.

Problem	C.o.I.	#Cells	#Steps	Input(MB)	#Vars	#Clauses	ExTime
F^1+B		10	18	5	22173	280987	1m 48s
F^1+B	$\surd$	10	18	4	16773	241961	1m 59s
$F^1+B+10L$		10	18	5	22173	296605	2m 05s
$F^1+B+10L$	$\surd$	10	18	4	16773	257561	1m 40s
$F^1+B+30L$		10	18	5	22173	330962	54s
$F^1+B+30L$	$\surd$	10	18	4	16773	288791	59s
$100F^1+B$		10	18	4	22173	281283	2m20s
$100F^1+B$	$\surd$	10	18	4	16773	241961	2m

C.o.I.=Reduction of the SAT-formula via the Cone of Influence

Table 5. Experiments on F^1-like problems.

configuration. The execution time further improves when forcing zChaff to select the variables occurring in F with the trick discussed at the beginning of this section. Specifically, when duplicating the last step of the formula $Ev^{\mathcal{A}}$ zChaff takes $35s$ to solve the same problem (see prop. $F+B+2L$ in Table 4). Similar results are obtained by simply copying F 10 times in the input formula given to zChaff (see prop. $10F+B$ in Table 4). Tuning the heuristics might be difficult (without modifying the code of zChaff) as shown by further experiments (like $100F+B$) where the performance gets worse again.

F^1 Properties In order to study the effectiveness of the cone of influence reduction we have also considered $F^{-1}+B$ properties as shown in Table 5. The use of this reduction alone does not improve the execution time much (see F^1+B with and without C.o.I. in Table 5). However, when coupled with the n-copy of the last step of $Ev^{\mathcal{A}}$ (to force the selection of variables in F) then it seems to work (see $F^1+B+10L$ with and without C.o.I. in Table 5). Copying the formula F does not seem to work well in this examples. Again tuning the parameters used in heuristic like the number of copies n in $\ldots+nL$ might be difficult by simply using zChaff as a black box.

Problem	#Cells	#Steps	#Vars	#Clauses	Input(MB)	ExTime
I	10	18	738044	12534004	237.00	5s
I	15	1	61564	1044495	17.26	5s
I	15	2	123064	2088990	36.02	9s
I	15	3	123064	3133485	56.23	14s
I	15	10	651064	10445014	197.70	48s
I	15	15	922564	15667489	299.00	1m54s
I	15	17	1045564	17756479	341.00	2m00s
I	15	20	1230064	20889964	410.41	ABORT

Table 6. Experiments on a CA with 4096 table rows.

5.4 Other Examples

In the last series of experiments we have randomly generated a CA with more than 4000 table rows and tested on it I-properties of increasing size. The aim here was to reach the limit of zChaff w.r.t. number of variables and clauses generated by the encoding on an easy problem. As shown in Table 6, zChaff gets in trouble when the formula has more than one million variables and about 20 millions of clauses (15 cells, 20 steps), while it can handle problems with 17 millions clauses (15 cells, 17 steps). This kind of analysis can be useful to evaluate the size of CAs we can handle with non-specialized SAT solvers.

6 Conclusions and Related Work

Although several CAs programming and simulation tools have been developed (see e.g. the survey of [22]), we are not aware of general frameworks for performing qualitative analysis of CAs automatically. In this paper we have proposed a SAT-based methodology for attacking this problem. One of the advantages of the proposed method is that, once the encoding of the CA-evolution has been computed, several different reachability problems can be formulated as simple propositional queries to a SAT-solver. The formula encoding the evolution can then be reused in a modular way (we can shrink or extend it easily and attach to it different initial/final configurations). Hard problems like inverse reachability can be attacked then by using modern SAT-solvers like zChaff that seems to perform well on problems with millions of variables and clauses.

Although this seems a new approach for checking properties of CAs, SAT technology is widely used for computer aided verification of hardware and software design. As an example, tools for Bounded Model Checking like nuSMV [1] automatically generate an *unfolding* of the transition relation of a high level description of a reactive system and then use a SAT-solver to test LTL properties. The reason why we did not resort to existing SAT-based verification tools like nuSMV is that we wanted to have complete control over the SAT-formula generated by the encoding of a CA-evolution. This way we were able to test different

kind of properties and heuristics on zChaff directly.

In our preliminary experiments we have obtained interesting results for CAs of reasonable size (e.g. 70 cells, 140 steps, 120 rules). We believe that it might be possible to manage larger problems by a specialization of the SAT-solving algorithm (and, especially, of its heuristics) that could benefit from structural properties of CAs. This might be an interesting future direction for our research.

References

1. A. Biere, E.M. Clarke, R. Raimi, Y. Zhu. *Verifiying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs*, CAV '99, Lecture Notes in Computer Science, 60-71, 1999.
2. E. Burks. *Essays on Cellular Automata*, University of Illinois Press, 1972.
3. E.F.Codd. *Cellular Automata*, Academic Press, New York, 1968.
4. M. D'Antonio. *Analisi SAT-based di Automi Cellulari*, Tesi di laurea, Dip. di Informatica e Scienze dell'Informazione, Università di Genova, Marzo 2004.
5. M. Davis, H. Putnam. *A computing procedure for quantification theory*, Journal of the Association for Computing Machinery, 1960.
6. F. Green. *NP-Complete Problems in Cellular Automata*, Complex Systems, 1:453-474, 1987.
7. J. Groote, J. Warners *The propositional formula checker HeerHugo*, Journal of Automated Reasoning, 24(1-2):101-125, 2000.
8. The ICS home page: http://www.icansolve.com
9. J. Kari. *Cryptosystems based on reversible cellular automata*, Preprint, 1992.
10. J. Kari. *Reversability of 2D cellular automata is undecidable*, Physica, Vol. D 45:379-385, 1990.
11. C. Langton. *Studying artificial life with cellular automata*, Physica D, 22:120-140, 1986.
12. J. Mazoyer. *A six-state minimal time solution to the firing squad synchronization problem*, Theoretical Computer Science, 50(2):183-240, Elsevier, 1987.
13. E.F. Moore. *Sequential machines* in: E.F. Moore *Selected Papers*, Addison-Wesley, Reading, 1964.
14. M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik. *Chaff: Engineering an Efficient SAT Solver*, Proceedings of the 38th Design Automation Conference (DAC'01), 2001.
15. D. A. Plaisted and S. Greenbaum. A Structure Preserving Clause Form Translation. Journal of Symbolic Computation, 2(3):293–304, 1986.
16. The SIMO web page: http://www.mrg.dist.unige.it/ sim/simo/
17. K. Sutner. *On the computational complexity of finite cellular automata*, JCSS, 50(1):87-97, 1995.
18. J. von Neumann. *Theory of Self-Reproducing Automata*, University of Illinois Press, edito e completato da A.W. Burks, 1966.
19. S. Wolfram. *Universality and complexity in cellular automata*, Physica D, 10:1-35, 1984.
20. S. Wolfram.*Computation Theory of Cellular Automata*, Communications in Mathematical Physics, 96:15-57, November 1984.
21. S. Wolfram. *Theory and Applications of Cellular Automata*, World Scientific, 1986.
22. T. Worsch. *Programming Environments for Cellular Automata*,Technical report 37/96, Universitat Karlsruhe, Fakultat fur Informatik, 1996.

Uniform relational frameworks for modal inferences*

A. Formisano[1], E. G. Omodeo[1], E. S. Orłowska[2], and A. Policriti[3]

[1] Dip. di Informatica, Univ. dell'Aquila. {formisano|omodeo}@di.univaq.it
[2] Instytut Łączności, Warsaw, Poland. orlowska@itl.waw.pl
[3] Dip. di Matematica e Informatica, Univ. di Udine. policriti@dimi.uniud.it

Abstract. The design and the implementation of variable-free deductive frameworks (ultimately based on Tarski's arithmetic of dyadic relations) for aggregate theories, strongly rely on the availability of suitable pairing notions.
Thanks to a set-theoretical treatment of modalities, we show that even in the case of modal propositional logics, an appropriate pairing notion can be introduced and exploited in order to devise an alternative approach to modal deduction.

Key words: *Modal logic*, *relational systems*, *translation methods*.

Introduction

In this paper we focus on a technical issue which plays a crucial role in algebraic formalization of set theory. The kind of formalization we have in mind sprouts from the historical line of work presented in the monograph [17]; but our contribution is much more specific, as it links with previous work on set-theoretic renderings of non-classical logics, and such renderings only presuppose very weak axioms concerning sets. It has been shown, in fact (cf. the *box-as-powerset*, in brief $\Box$-as-$\mathcal{P}$, translation of [4, 1]), that even a very weak theory can offer adequate means for expressing the semantics of modal systems of propositional logic, for investigating the issue of their first-order representability, and for automating inferences in non-classical contexts (cf. [10, pp. 478–481]). Unlike previous studies, where suitable weak set theories were formalized within ordinary quantified calculus (in one case, namely [14], an inferential system *à la* Rasiowa-Sikorski was developed), here we provide a variable-free, equational version of the target set-theoretic language. To compensate for the lack of variables and quantifiers, the equality construct will be used for comparing expressions designating global relations between sets rather than for comparing set-expressions.

Providing a suitable notion of *pairing* is the central issue of the proposal we put forward; in fact, pairing will be a key tool for obtaining, with minimal axiomatic commitment, the desired equational support for the said set-rendering

* This research benefited from collaborations fostered by the European action COST n.274 (TARSKI, see http://www.tarski.org).

of modal logics specified *à la* Hilbert. Such a tool enables use of the Maddux-Monk-Tarski algorithm (cf. [17]) for restating any first-order sentence in three variables; so it paves the way to the equational treatment of non-classical logics. As the present paper will illustrate through various examples stemming from the general approach discussed in [2], the proposed pairing device can be built into a specialized variant of the $\Box$-as-$\mathcal{P}$ translation so as to drive it into a completely equational setting.

An opportunity for improvements would have been missed if we had simply proposed a montage of the $\Box$-as-$\mathcal{P}$ translation with Maddux' translation. To mention one thing, the first-order sentences resulting from $\Box$-as-$\mathcal{P}$ constitute a rather narrow sublanguage of first-order logic, whose 3-variable translation can easily be obtained by more direct means than by the general method. Moreover, in view of the special purpose of our translation, the first-order (set-theoretic) equivalent of a modal proposition can be left understood, and the two logical phases of the overall translation can be fused into a single process. The role of Maddux' translation, in this process, can be superseded by the graph-based approach developed in [2], whose techniques, however, are conservative and do not guarantee success in all circumstances; therefore, suitable extensions of that approach had to be devised to take full advantage of the special context of our discourse, where most of the relations entering into play are functions, and where conjugated projections (left and right inverses of the pairing function) are available.

As for the pairing function, none seems to be available under the very weak assumptions of the target set theory as originally proposed in [4, 1]. We must either add an explicit pair axiom, or design a suitable variant of the $\mathcal{P}$ operation (cf. Sec.1): in either case, a well-known device due to Kuratowski (whereby ordered pairs can be encoded in terms of nested unordered pairs) can be adopted.

This paper is 'twin', in a sense, of [14]. Both develop a theory-based approach, whose difference relative to a more typical calculus-based approach will be sketched in Sec.3. Suffice it to say, for the time being, that, typically, one assumes the universe of discourse to consist exclusively of the possible worlds of a frame; here, instead, the universe of discourse results from the amalgamation of all Kripkean frames and hence encompasses worlds and frames together (as well as other less relevant entities), all uniformly viewed as 'sets'. Under one important aspect this paper and [14] diverge, though: the Rasiowa-Sikorski system adopted there is particularly handy and human-oriented, whereas the equational kind of reasoning to which modal logics are reduced in this paper is intended to be mainly oriented to machine formula-crunching.

1 Background axiomatic aggregate theory

Consider the axiomatic theory Ω whose postulates are

$$\begin{aligned}
\forall y\,\forall x\,\exists n\,\forall v\big(\; v\in n \;\leftrightarrow\; & \,(v\in x \;\&\; v\in y)\;\big)\,,\\
\forall y\,\forall x\,\exists d\,\forall v\big(\; v\in d \;\leftrightarrow\; & \neg\,(v\in x\leftrightarrow v\in y)\;\big)\,,\\
\forall x\,\exists p\,\forall v\big(\; v\in p \;\leftrightarrow\; & \,(\forall u\in v)(u\in x)\;\big)\,.
\end{aligned}$$

By resorting to Skolem operators $\cap$, $\triangle$, and $\mathcal{P}$, and to the syntactic abbreviation

$$v \subseteq x \ \leftrightarrow_{\mathrm{Def}} \ (\forall\, u \in v)(u \in x)\,,$$

and leaving the quantifiers $\forall\, v$, $\forall\, x$, $\forall\, y$ tacit, we can recast the above axioms more perspicuously as follows:

$$\begin{array}{rcl} v \in x \cap y & \leftrightarrow & (v \in x \quad \& \quad v \in y)\,, \\ v \in x \triangle y & \leftrightarrow & (v \in x \quad \leftrightarrow \quad \neg\, v \in y)\,, \\ v \in \mathcal{P}(x) & \leftrightarrow & v \subseteq x\,. \end{array}$$

One can view Ω as being an extremely weak theory of 'aggregates' which becomes a genuine set theory only after appropriate postulates, such as the *extensionality* axiom

$$(x \subseteq y \ \& \ y \subseteq x) \ \rightarrow \ x = y$$

and the *(unordered) pair* axiom

$$v \in \{x, y\} \ \leftrightarrow \ (v = x \vee v = y)\,,$$

are added to it. On the other hand Ω is known to be, already as it stands, an ideal target first-order theory into which to translate mono-modal systems of propositional logic uniformly (cf. [3, Chapter 12]): in the translation, the converse $\ni$ of membership acts as a relation which includes immediate accessibility between possible worlds; accordingly, $\cap$ and $\triangle$ play the role of the classical connectives $\&$, $\oplus$—conjunction and exclusive disjunction—, and $\mathcal{P}$ corresponds to the necessity operator $\Box$.

From the standpoint of this '$\Box$-*as*-$\mathcal{P}$' translation, the weakness of Ω is a virtue rather than a defect. As a matter of fact, if the extensionality axiom were included in Ω, this would set an undesirable limitation to its usability in the study of non-classical logics; and a similar objection can be raised against postulates entailing the well-foundedness of membership. Certain enrichments of Ω with new postulates, e.g. the addition of the pair axiom, do not jeopardize applicability of the $\Box$-as-$\mathcal{P}$ translation method; nevertheless such enrichments appear to be unjustified unless they are shown to yield some technical—perhaps computational—advantages.

The important result summarized in this paragraph, established by Alfred Tarski over half century ago and later improved by J. Donald Monk and Roger Maddux (cf. [17, Chapter 4]), seems to favor the addition of the pair axiom to Ω. An effective procedure exists for reducing each sentence of the language underlying any first-order theory of membership which includes the pair axiom to an equivalent sentence involving three variables only; furthermore, this procedure enables global translation of such a theory into a purely equational extension of the *arithmetic of dyadic relations*, which is an algebraic theory owning an "almost embarrassingly rich structure" [12]. Thanks to such a translation, we might gain better service from today's theorem provers run in autonomous mode: in our own experience, these generally demonstrate higher performances when confronted with purely equational theories than with theories which more fully

exploit the symbolic first-order apparatus [8, 9]. By translating modal axioms and theses into an equational variant of Ω, we could hence best benefit of current proof technology.

We will propose below an even less committing way of reducing (a variant Ω' of) Ω to the arithmetic of relations, taking advantage of the fact that the above cited Tarski-Maddux' result holds also for theories where an analogue of the pair axiom, of the form

$$\forall y\,\forall x\,\exists q\,\forall v\big(\,v \;\mathsf{in}\; q \;\leftrightarrow\; (v = x \vee v = y)\,\big)\,,$$

can be derived from the axioms. The only requirement, in regard to this, is that "v in q" be a formula which involves three variables altogether and has v and q as its sole free variables. To achieve our translation purpose, we just have to retouch the one axiom which characterizes the 'powerset' operator $\mathcal{P}$ so that it behaves more naturally when the extensionality axiom is missing. Our proposed replacement for the third axiom of Ω is the sentence

$$\forall x\,\exists p\,\forall v\big(\,v \in p \;\leftrightarrow\; (v = x \vee v \subset x)\,\big)\,,$$

where

$$v \subset x \;\;\leftrightarrow_{\mathrm{Def}}\; \neg\,(v \subseteq x \rightarrow x \subseteq v)$$

(that is, $v \subset x$ holds if and only if every element of v belongs to x whereas x has some element not belonging to v). Under this revised axiom, even without extensionality axiom, it is clear that exactly one p, let us call it $\tilde{\mathcal{P}}(x)$, corresponds to each x so that the elements of p are precisely x and all of its strict subsets $v \subset x$. Likewise, to any q there corresponds at most one a such that q max a holds, where

$$q \;\mathsf{max}\; a \;\;\leftrightarrow_{\mathrm{Def}}\; \big(a \in q \;\&\; q \subseteq \tilde{\mathcal{P}}(a)\big)\,;$$

but, unlike $\tilde{\mathcal{P}}$ which is total, max is a *partial* function of its first operand.

In our revised version Ω' of Ω, one can conceive an analogue of the unordered pair $\{a, b\}$ to be $\tilde{\mathcal{P}}(a)$ when $a = b$ and to have the same elements as $\tilde{\mathcal{P}}(a)\triangle\tilde{\mathcal{P}}(b)$ when $a \neq b$. (Out of such 'unordered pairs', one can proceed to construct 'ordered pairs' analogous to Kazimierz Kuratowski's pairs $\langle a, b\rangle =_{\mathrm{Def}}\big\{\{a, b\}, \{a, a\}\big\}$, and these will behave as desired, but let us avoid a discussion on this point taking it for granted [6].) With this rationale in mind, we can characterize as follows a 'pseudo-membership' which meets the formal analogue seen above of the pair axiom:

$$b \;\mathsf{in}\; q \;\;\leftrightarrow_{\mathrm{Def}}\; \left(b \in q \;\&\; \big(\neg\,\exists\, d \in q\big)\big(b \subset d\big)\right) \vee \exists\, a\Bigg(q \;\mathsf{max}\; a \;\&\; b \subset a \;\&\; \forall\, d\Big(d \in q \leftrightarrow \big(d \in \tilde{\mathcal{P}}(a) \;\&\; \neg\, d \in \tilde{\mathcal{P}}(b)\big)\Big)\Bigg)\,.$$

To see that in can be specified in three variables, it suffices to observe that since

$$\ni \;=_{\rm Def}\; \in^{\smile} \qquad\qquad \supseteq \;=_{\rm Def}\; \overline{\not\ni\in}$$
$$\supset \;=_{\rm Def}\; \supseteq \cdot \ni\!\notin \qquad\qquad \subset \;=_{\rm Def}\; \supset^{\smile}$$
$$\tilde{\mathcal{P}} \;=_{\rm Def}\; \in - \supset\!\notin - (\bar{\iota} \cdot \not\supset) \in \qquad\qquad \mu \;=_{\rm Def}\; \tilde{\mathcal{P}} \supseteq \cdot \in$$
$$\mathsf{in} \;=_{\rm Def}\; \Big(\in - \subset\in\Big) \sqcup \Big(\subset \mu - \tilde{\mathcal{P}} \ni\in - \mathbb{1}\,(\in - \in \tilde{\mathcal{P}}^{\smile} \mu) - \tilde{\mathcal{P}} \not\ni (\in \tilde{\mathcal{P}}^{\smile} \mu - \in)\Big)$$
$$\gamma \;=_{\rm Def}\; (\,\mathsf{in} - \bar{\iota}\,\mathsf{in}\,)\,\mathsf{in} \qquad\qquad \mathsf{syq}(P,\,Q) \;=_{\rm Def}\; \overline{P^{\smile}\;\overline{Q}} \cdot \overline{\overline{P}^{\smile}\;Q}$$
$$\lambda \;=_{\rm Def}\; \gamma - \bar{\iota}\,\gamma \qquad\qquad \varrho \;=_{\rm Def}\; \mathsf{in}\,\mathsf{in} - \bar{\iota}\,(\,\mathsf{in}\,\mathsf{in} - \lambda\,)$$

$$\tilde{\mathcal{P}}\,\mathbb{1} = \mathbb{1} \qquad\qquad \lambda\,\varrho^{\smile} = \mathbb{1}$$
$$\mathbb{1} = \mathbb{1}\;\mathsf{syq}(\in,\, \in\lambda \cdot \in\varrho) \qquad\qquad \mathbb{1}\;\mathsf{syq}(\in,\, \in\lambda + \in\varrho) = \mathbb{1}$$

Fig. 1. Specification of Ω' in the arithmetic of dyadic relations

max is single-valued, the *definiens* of the predicate in can be rewritten as follows:

$$\Big(b \in q \;\&\; (\neg\, \exists\, d \in q)\big(b \subset d\big)\Big) \;\vee\; \Big(\exists\, d\,(q\;\mathsf{max}\;d \;\&\; b \subset d)$$
$$\& \;\forall\, d\Big(d \in q \leftrightarrow \Big(\big(\neg\, d \in \tilde{\mathcal{P}}(b)\big) \;\&\; \exists\, b\big(q\;\mathsf{max}\;b \;\&\; d \in \tilde{\mathcal{P}}(b)\big)\Big)\Big)\Big).$$

Given the above definition of in, let us consider the following pair of relations:

$$a\,\lambda\,q \;\leftrightarrow_{\rm Def}\; a\,\gamma\,q \;\&\; \forall\, z\,(z\,\gamma\,q \rightarrow z = a),$$
$$b\,\varrho\,q \;\leftrightarrow_{\rm Def}\; \exists\, w\,(b\;\mathsf{in}\;w \;\&\; w\;\mathsf{in}\;q) \;\&\; \forall\, z\,\big(\exists\, v\,(z\;\mathsf{in}\;v \;\&\; v\;\mathsf{in}\;q) \rightarrow z = b \,\vee\, z\,\lambda\,q\big),$$

where $x\,\gamma\,y \;\leftrightarrow_{\rm Def}\; \exists\, z\,\big(x\;\mathsf{in}\;z \;\&\; z\;\mathsf{in}\;y \;\&\; \forall\, w\,(w\;\mathsf{in}\;z \rightarrow w = x)\big)$.
Since in is expressed in three variables, it is easy to verify that both λ and ϱ are defined in terms of 3-variable sentences. Notice that, thanks to their definitions, both λ and ϱ are functions of their second operands. According to [17] (see also [6] and Sec.5 below), the availability of such a pair of relations allows one to recast the pairing axiom in three variables, as follows:

$$\forall\, x\, \forall\, y\, \exists\, q\,(x\,\lambda\,q \;\&\; y\,\varrho\,q).$$

The equational rendering of this axiom will be part of the equational specification of Ω', as shown in Figure 1 (where an equational rendering of λ and ϱ is also shown).

At present we do not know whether Ω', without further axioms, can supersede Ω as target first-order theory into which modal propositional logics can be translated naturally. In fact, the proof of the theorem which states the adequacy of the $\Box$-as-$\mathcal{P}$ translation into Ω (cf. [4, Sec.3]) cannot be carried over in an obvious manner to our slightly altered context.

2 Arithmetic of homogeneous dyadic relations

We got *in medias res*.[4] The algebraic specification of Ω' just reached calls for a quick flash back into our own variant of the algebraic form of logic historically

[4] Directly to the heart of the tale.

developed by Charles Sanders Peirce, Ernst Schröder, and Alfred Tarski [17], to recall a few.

Our intended *universe of discourse* is a collection $\Re$ of dyadic relations over a non-null domain $\mathcal{U}$. We assume that the *top* relation $\bigcup\Re$, and the *diagonal* relation, consisting of all pairs $\langle u, u\rangle$ with u in $\mathcal{U}$, belong to this universe, which is also closed under the *intersection*, *symmetric difference*, *composition*, and *conversion* operations. Within our symbolic algebraic system, the constants $\mathbb{1}$ and ι designate the top and the diagonal relation; moreover $\cdot$ and $+$ designate intersection and symmetric difference; composition is represented by simple juxtaposition; and conversion by the monadic operator $^{\smile}$. These operations are interpreted as follows, where for any relational expression R we are indicating by $R^{\Im}$ the relation (over $\mathcal{U}$) designated by R:

- $P^{\smile}$ designates the relation consisting of all pairs $\langle v, u\rangle$ with $\langle u, v\rangle$ in $P^{\Im}$;
- $P\,Q$ designates the relation consisting of all pairs $\langle u, w\rangle$ such that there is at least one v for which $\langle u, v\rangle$ and $\langle v, w\rangle$ belong to $P^{\Im}$ and to $Q^{\Im}$, respectively;
- $P{\cdot}Q$ designates the relation consisting of all pairs $\langle u, v\rangle$ which simultaneously belong to $P^{\Im}$ and to $Q^{\Im}$;
- $P{+}Q$ designates the relation consisting of all pairs $\langle u, v\rangle$ which belong either to $P^{\Im}$ or to $Q^{\Im}$ but do not belong to both of them.

Designations for customary operations over relations can be introduced through shorthand definitions, e.g. as follows:

$$\overline{Q} =_{\mathrm{Def}} Q{+}\mathbb{1}, \qquad \emptyset =_{\mathrm{Def}} \overline{\mathbb{1}},$$
$$P{-}Q =_{\mathrm{Def}} P{\cdot}\overline{Q}, \qquad P{\sqcup}Q =_{\mathrm{Def}} \overline{\overline{Q}{-}P}.$$

We will sometimes exploit alternative notation for the complement operation and for equations of a special kind:

$$\not{Q} =_{\mathrm{Def}} \overline{Q}, \qquad P{\sqsubseteq}Q \;\leftrightarrow_{\mathrm{Def}}\; P{-}Q{=}\emptyset.$$

Further abbreviations can be introduced at will, as shown for example in Figure 1 by the introduction of the *symmetric quotient* operation syq (cf. [16, pp. 19–20, 71ff]). As concerns *priorities*, we assign to $^{\smile}$, composition, $\cdot, +, -$, and $\sqcup$ decreasing cohesive powers; moreover, all dyadic constructs, namely $\cdot, +, -, \sqcup$, and composition, are assumed to associate to the left.

Figure 2 displays an axiomatic presentation of the *arithmetic of relations*, which also encompasses the standard equational inference rules. We regard these axioms as *logical* ones, because they form, in a sense (together with the inference rules), a *calculus* on top of which one can build purely equational theories, such as the Ω' theory introduced above. Specific *theories* will talk about special relations—the one designated by $\in$, in the case of Ω'—, which they constrain to comply with *proper* axioms—e.g. the ones shown in Figure 1. A theory characterizes peculiar domains endowed with special relations. For example, the axioms of Ω' imply that the domain $\mathcal{U}$ is infinite and has a considerable amount of structure. This actually reflects our willingness to impose enough structure on $\mathcal{U}$ that it can be regarded as an amalgamation of all Kripke frames upon which the '*possible-world*' semantics of modal propositional logics is based.

$$
\begin{array}{cc}
P \cdot Q = Q \cdot P & P \cdot (Q + R) + P \cdot Q = P \cdot R \\
(P \cdot Q) \cdot R = P \cdot (Q \cdot R) & (P + Q) + R = P + (Q + R) \\
\mathbb{1} \cdot P = P & \iota \;\; P = P \\
(P \;\; Q) \;\; R = P \;\; (Q \;\; R) & (P \sqcup Q)\, R = (Q\, R \sqcup P\, R) \\
(P \cdot Q)^{\smile} = Q^{\smile} \cdot P^{\smile} & (P\, Q)^{\smile} = Q^{\smile}\, P^{\smile} \\
P^{\smile\smile} = P & Q \cdot \big((Q \;\; P + \mathbb{1}) \;\; P^{\smile}\big) = \emptyset
\end{array}
$$

Fig. 2. Logical axioms of the arithmetic of dyadic relations

3 Direct relational translations of modal logics

A vast literature exists on how to translate non-classical propositional logics into first-order predicate calculus and how to exploit such translations for automated non-classical reasoning, cf. e.g. [18].

Consider, for example, the following translation mapping (essentially the one proposed in [15]), which associates a relational expression $t(\varphi)$ with each modal propositional sentence φ :

- $t(p_i) \;\; =_{\mathrm{Def}} \;\; p'_i \;\; \mathbb{1}$, where p'_i is a relational variable uniquely corresponding to p_i , for every propositional variable p_i;
- $t(\neg\,\psi) \;\; =_{\mathrm{Def}} \;\; \overline{t(\psi)}$, for every propositional sentence ψ ;
- $t(\psi \;\&\; \chi) \;\; =_{\mathrm{Def}} \;\; t(\psi) \cdot t(\chi)$, for all propositional sentences ψ, χ ;
- $t(\Diamond\,\psi) \;\; =_{\mathrm{Def}} \;\; \mathsf{r} \;\; t(\psi)$, where r is a constant designating the *accessibility relation* between possible worlds, for every propositional sentence ψ .

Of course the connectives $\rightarrow, \vee, \leftrightarrow, \oplus, \Box$ can be handled similarly, via reductions to $\neg$, $\&$, $\Diamond$. It is also plain that $t(\varphi)$ designates a *right-ideal* relation Φ; namely, one which satisfies the equation $\Phi\;\mathbb{1} = \Phi$.

Assume that we have been able to capture the semantics of a specific logic $\mathbb{L}$ by means of a system of relational equations $\mathcal{E}(\mathbb{L})$ *involving* r *alone*: then we can, in place of the problem of establishing whether or not $\models_{\mathbb{L}} \vartheta$ (where ϑ is any modal formula), address the equivalent problem of establishing whether or not $\mathcal{E}(\mathbb{L}) \vdash t(\vartheta){=}\mathbb{1}$. One can see the condition $\mathcal{E}(\mathbb{L}) \vdash t(\vartheta){=}\mathbb{1}$ as a statement referring to first-order predicate calculus, because we can re-express the (relatively unusual) relational constructs it involves in terms of individual variables and quantifiers. Indeed, we can rewrite $P{=}\mathbb{1}$ as $\forall x \forall y (\, x\, P\, y\,)$, $x P \cdot Q y$ as $x\, P\, y \;\&\; x\, Q\, y$, $x P Q y$ as $\exists\, z (x\, P\, z \;\;\&\;\; z\, Q\, y)$, etc.

This approach is viable for a wide spectrum of non-classical logics, including some which are directly characterized in terms of semantic constraints which the accessibility relation is subject to, for example *extensionality*

$$\mathsf{syq}(\mathsf{r},\ \mathsf{r}) \sqsubseteq \iota$$

(a property of contact relations used in spatial reasoning, cf. [5]). On the other hand, there exist modal logics very simply characterizable via Hilbert-like axioms

which *do not* admit a first-order correspondent, one such being the single-axiom Löb logic

$$\Box(\Box p \rightarrow p) \rightarrow \Box p\,.$$

For such defective logics a correspondent can always be found in *second-order* predicate calculus, which unfortunately is not complete (no matter what recursive set of logical axioms is chosen, cf. [13]).

An alternative possibility, which we will discuss in the ongoing, is to take a first-order *theory* (as opposed to the mere calculus) as the target formalism for the translation. A most natural choice, when $\mathbb{L}$ is given by a finite conjunction α of Hilbert-like axioms, is to refer to the arithmetic $\mathcal{RA}$ of relations (see Figure 2): we can then take $\mathcal{E}(\mathbb{L})$ to be $t(\alpha){=}\mathbb{1}$, and try to see whether $\models_{\mathbb{L}} \vartheta$ by checking whether $\mathcal{RA}$ & $\big(t(\alpha){=}\mathbb{1}\big) \vdash t(\vartheta){=}\mathbb{1}$. This approach would lack completeness too (since $\mathcal{RA}$ does not axiomatize the whole variety of representable relation algebras), despite being sound, efficient (as automatic theorem-proving in purely equational contexts is faster than for full first-order logic), and able to provide answers in common cases.

From its very origin, the theory Ω was devised with the aim that it should retain enough of the power of second-order predicate calculus to make recourse to second order useless, while ensuring completeness. In its original form, however, Ω was not immediately amenable to an equational extension of $\mathcal{RA}$; hence it did not enable emulation of non-classical reasoning in the purely equational part of the arithmetic of relations; now, thanks to the availability of conjugated projections, it will.

4 Modal logics and the arithmetic of relations

Let us recall from [4], [1] (see also [3, Chapter 12]) that there is a translation $\varphi \mapsto \varphi^\star$ of modal propositional sentences into set-terms of Ω which enjoys the following properties:

- A sentence schema $\varphi \equiv \varphi[p_1, \ldots, p_n]$ built from n distinct propositional meta-variables p_i becomes a term $\varphi^\star \equiv \varphi^\star[f, x_1, \ldots, x_n]$ involving $n+1$ distinct set-variables, one of which, f, is meant to represent a generic frame.
- If $\varphi^\star \equiv \varphi^\star[f, \boldsymbol{x}]$ and $\psi^\star \equiv \psi^\star[f, \boldsymbol{y}]$ result from propositional schemata φ, ψ, then the biimplication

$$\psi \models_{\mathsf{K}} \varphi \Leftrightarrow \Omega \vdash \forall f\big(\mathsf{is_frame}(f) \wedge \forall \boldsymbol{y}(f \subseteq \psi^\star) \rightarrow \forall \boldsymbol{x}(f \subseteq \varphi^\star)\big)$$

  holds, where K is the minimal modal logic and $\mathsf{is_frame}(\,\cdot\,)$ characterizes those elements of the intended domain $\mathcal{U}$ which represent Kripke frames.

Hence, by combining this translation with a proof system for Ω, one achieves a proof system which can be exploited to semi-decide any finitely axiomatized mono-modal propositional logic (and even, in favorable cases, to decide it). We would like to now tune the same translation for our relational theory Ω', with the additional advantage that the target formalism would be a purely equational

one in the case at hand; however, due to the difficulty mentioned at the end of Sec.1, in the rest of this paper we replace Ω' by the (equational) theory Ω'' consisting of the axioms of Ω' plus the axiom

$$\mathcal{P}\,\mathbb{1} = \mathbb{1}, \quad \text{where} \quad \mathcal{P} =_{\mathrm{Def}} \mathsf{syq}(\subseteq, \in) \quad \text{and} \quad \subseteq =_{\mathrm{Def}} \overline{\ni \notin}\,.$$

We begin by specifying the monadic relation $\mathsf{is_frame}$, bearing in mind that since $\ni$ must act, when restricted to a frame, as the accessibility relation between worlds in that frame, a frame f must be a transitive set, i.e., it must satisfy $f \subseteq \mathcal{P}(f)$. This characterization turns out to be adequate to our purposes:

$$\mathsf{is_frame} =_{\mathrm{Def}} \iota \cdot \mathcal{P} \supseteq\,.$$

Then we must specify operations on $\mathcal{U}$ that correspond to the propositional constructs $\vee$, $\&$, and $\Box$. Here the rationale is that the term $\varphi[f, \boldsymbol{x}]$ into which one translates a propositional schema $\varphi^\star[\boldsymbol{p}]$ represents the collection of all worlds (in the frame f) where φ holds. We have announced already in Sec.1 that the natural counterparts of $\Box$ and $\&$ will be $\mathcal{P}$ and $\cap$: analogously, $\cup$ and $\setminus$ will act as counterparts of $\vee$ and $\not\rightarrow$, but since we have not introduced any of $\cap$, $\cup$, and $\setminus$ explicitly in Figure 1, we can fill this gap now by putting[5]

$$\cap =_{\mathrm{Def}} \mathsf{syq}(\in, \in\lambda \cdot \in\varrho)\,, \qquad \cup =_{\mathrm{Def}} \mathsf{syq}(\in, \in\lambda \sqcup \in\varrho)\,,$$
$$\setminus =_{\mathrm{Def}} \mathsf{syq}(\in, \in\lambda - \in\varrho)\,.$$

In the language underlying the Skolemized first-order version of Ω'', it is quite straightforward to define the mapping $\varphi[\boldsymbol{p}] \mapsto \varphi^\star[f, \boldsymbol{x}]$ of sentence schemata into terms:

$$p_i^\star =_{\mathrm{Def}} x_i\,, \qquad (\neg\psi)^\star =_{\mathrm{Def}} f \setminus \psi^\star\,,$$
$$(\psi \rightarrow \chi)^\star =_{\mathrm{Def}} (\neg\psi)^\star \cup \chi^\star\,, \qquad (\Box\psi)^\star =_{\mathrm{Def}} \mathcal{P}(\psi^\star)\,.$$

(of course we can also handle the connectives $\&$, $\vee$, $\leftrightarrow$, $\oplus$, $\Diamond$ via reductions to $\neg$, $\rightarrow$, $\Box$). This is called the *$\Box$-as-$\mathcal{P}$ translation.* Assuming that we manage to specify the relation $f \not\subseteq \varphi^\star[f, \boldsymbol{x}]$ between a frame f and a tuple $\boldsymbol{x}$ by means of a relational expression $\widehat{\varphi}$, then the translation of $\Psi \vdash_{\mathsf{K}} \varphi$ (where Ψ stands for a finite collection of sentence schemata) into a derivability problem regarding the (quantifier-free, purely equational) relational version of Ω'' will be

$$\Omega'' \;\vdash\; \mathsf{is_frame} \cdot \widehat{\varphi}\,\mathbb{1} \sqsubseteq \widehat{\&\Psi}\;\mathbb{1}\,.$$

5 Compliance of a frame with a modal schema

How can we specify algebraically the relation (complement of the $\widehat{\varphi}$ just introduced) which holds between a frame f and a tuple $\boldsymbol{x}$ when $f \subseteq \varphi^\star[f, \boldsymbol{x}]$? A

[5] Needless to say, we must distinguish between $\cdot, +, -, \sqcup, \sqsubseteq$, which operate on relations, and corresponding constructs $\cap, \triangle, \setminus, \cup, \subseteq$, which operate on $\mathcal{U}$.

crucial observation is that within the relational version of Ω'' one can derive equations

$$L^{\smile}\,L \sqsubseteq \iota\,, \qquad R^{\smile}\,R \sqsubseteq \iota\,, \qquad L^{\smile}\,R = \mathbb{1}\,, \qquad \iota \sqsubseteq L\,\mathbb{1}\,R^{\smile}\,,$$

for suitably chosen relational expressions L, R. For example, we can take

$$L =_{\text{Def}} \lambda^{\smile} \sqcup (\iota - \lambda^{\smile}\,\mathbb{1})\,, \qquad R =_{\text{Def}} \varrho^{\smile} \sqcup (\iota - \varrho^{\smile}\,\mathbb{1})\,.$$

This remark immediately discloses the possibility of translating the full-fledged language of first-order Ω'' into the language of the arithmetic of relations: we can rely, for that, on a fundamental algorithm due to R. Maddux and explained in [17, Sec. 4.4]. Notice that the very fact that we can speak of *tuples* belonging to $\mathcal{U}$ relies on the availability of L, R. As a matter of fact, we can view each element a of $\mathcal{U}$ as a pair $\langle b, c\rangle$ whose components fulfill $aL^{\Im}b$ and $aR^{\Im}c$; accordingly, since we can decompose c in the same fashion, any element of $\mathcal{U}$ encodes a tuple of any desired length. On the other hand, $L^{\smile}\,R = \mathbb{1}$ implies that we can assemble a pair $\langle b, c\rangle$ from any two elements b, c of $\mathcal{U}$, and hence we can form an n-tuple with given components $b_1, \ldots, b_n$ for any finite number n.

We are not so much interested in translating the full first-order language into the equational one as we are in translating formulas $f \subseteq \varphi^{\star}[f, \boldsymbol{x}]$: these are, in fact, the ones we really need in order to carry out the translation of any sentence of modal propositional logic into relational Ω''. To this end, we can resort to a more direct graph-based translation approach explained in [2], which has been implemented by means of a tool for algebraic graph transformation named Agg (acronym for 'Attribute Graph Grammars'), developed at the TU, Berlin [11, 7].

Figure 3 displays the graph representation of $f \subseteq \varphi^{\star}[f, \boldsymbol{x}]$, when φ is one of the following modal sentences (x, y correspond to p and q, respectively):

S4: $\Box p \rightarrow \Box\Box p$;
Löb: $\Box(\Box p \rightarrow p) \rightarrow \Box p$;
K: $\Box(p \rightarrow q) \rightarrow (\Box p \rightarrow \Box q)$.

When fed with one of these three graphs, the thinning algorithm of [2, p. 455] terminates in a dead end. Figure 4 displays two of the resulting irreducible graphs, obscuring orientation and labels of the edges. Figures 5, 6, and 7 show, on a couple of examples, how to take advantage of the available 'projections' $\lambda^{\smile}$, $\varrho^{\smile}$ to bring an irreducible graph to a form whence the thinning algorithm can resume its job and terminate successfully, with the desired relational translation. (Notice that, the last graph of Figure 7 can be further reduced as done for the example in Figure 5.)

The key point is that, thanks to the availability of a pair of projections, $\lambda^{\smile}$ and $\varrho^{\smile}$, the thinning algorithm can be enriched with a new graph-rewriting rule:

PAIR-ENCODING rule. Let L and R be a pair of conjugated projections. Let ν' and ν'' be two nodes in a graph. Then introduce a new (bound) node ν together with two labelled edges $[\nu, L, \nu']$ and $[\nu, R, \nu'']$.

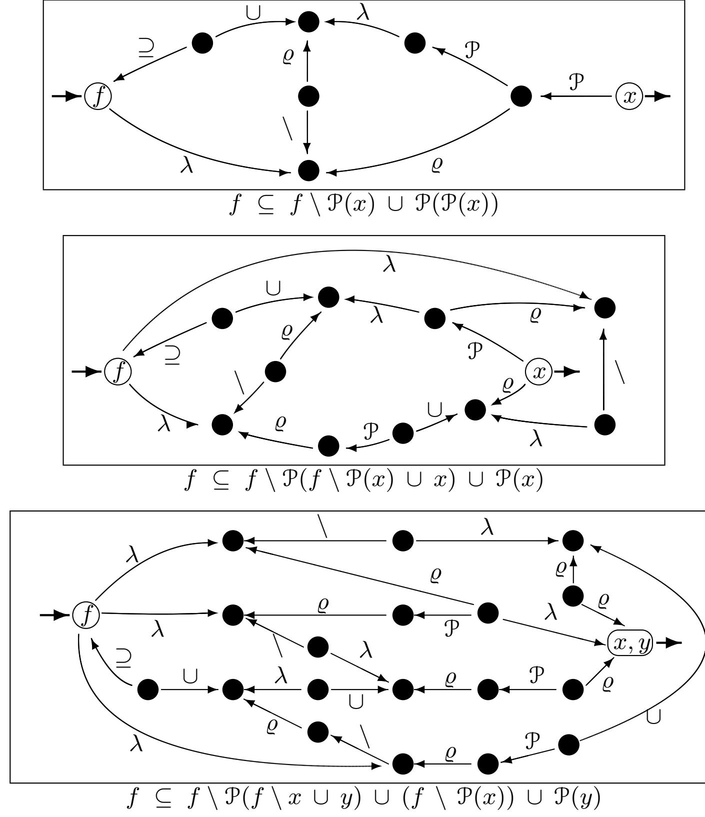


Fig. 3. Graph renderings of **S4**, **Löb**, and **K**

Such a rule is extremely general and, in principle, there is no restriction on its applicability. Clearly, it is convenient to exploit this rule within a strategy that relates its application to the firing of other rules of the thinning algorithm. In particular, since L and R are single-valued, the following rewriting-rule (cf. [2]) becomes crucial in order to bring to end the graph-rewriting process:

FUNCTIONALITY rule. Let $[\nu, P, \nu']$ be a labelled edge such that P is single-valued and let $[\nu', Q, \nu'']$ be another edge, with $\nu \neq \nu''$. Then the edge $[\nu', Q, \nu'']$ is removed and the new edge $[\nu, P\,Q, \nu'']$ is added to the graph. (If the graph contains another edge between ν and ν'' labelled S, then a fusion is made and the new edge will be labelled $P\,Q\cdot S$.)

Conclusions

In this paper we moved the first step toward the realization of an alternative deductive framework for non-classical logics based on pure equational reasoning.

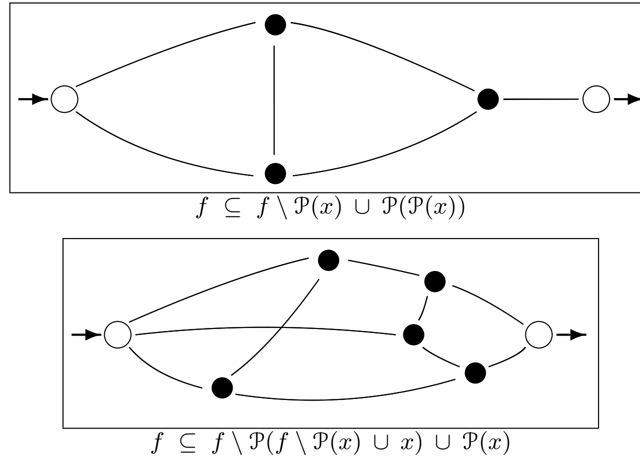


Fig. 4. Irreducible graphs resulting from **S4** and from **Löb** (cf. Figure 3)

We demonstrated how to profitably combine the experiences issuing from two apparently weakly-related streams of research: We (re-)forged a set-theoretical approach to the deduction problem in modal theories, within a purely equational deductive framework designed for set-reasoning and ultimately based on Tarski's relation algebra. To this aim we exploited the very same techniques developed for equational re-engineering of various aggregate theories [8, 6]. At the same time, this paper presents a new significant improvement of the translation technique proposed in [2] in order to compile first-order sentences into the algebra of relations.

Experimental activities with a state-of-the-art theorem-prover such as Otter [8] seem to indicate that equational formulations of aggregate theories can favorably compete with more conventional first-order formulations. Through the proposed equational rendering of a Ω'', thanks to the $\Box$-as-$\mathcal{P}$ translation, modal propositional logics are amenable to the jurisdiction such equational automated reasoning methods. Such a result emphasizes, once more, the expressive and deductive power of Tarski's algebraic theory of relations.

References

[1] J. F. A. K. van Benthem, G. D'Agostino, A. Montanari, and A. Policriti, Modal deduction in second-order logic and set theory-I, *J. of Logic and Computation*, **7**(2):251–265, 1997.

[2] D. Cantone, A. Formisano, E. G. Omodeo, and C. G. Zarba. Compiling dyadic first-order specifications into map algebra, *Theoretical Computer Science*, **293**(2):447–475, 2003.

[3] D. Cantone, E. G. Omodeo, and A. Policriti. *Set Theory for Computing – From decision procedures to declarative programming with sets.* Monographs in Computer Science, Springer, 2001.

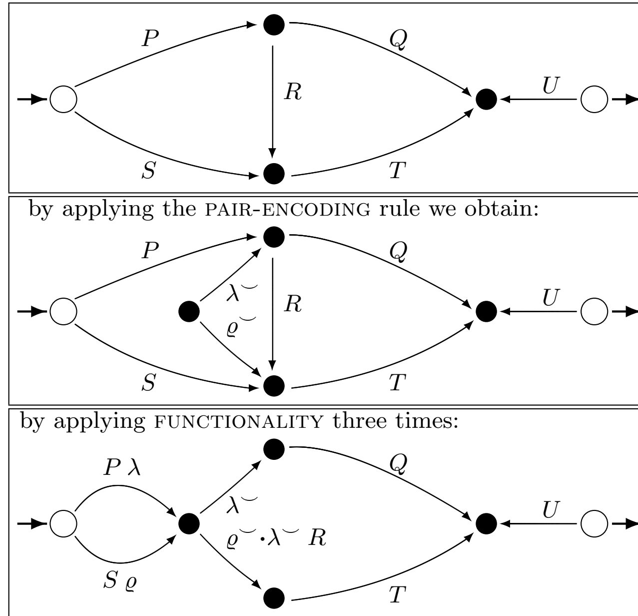


Fig. 5. Rehabilitation of an irreducible graph (cf. the first graph in Figure 4)

[4] G. D'Agostino, A. Montanari, and A. Policriti, A set-theoretic translation method for polymodal logics, *J. of Automated Reasoning*, **3**(15):317–337, 1995.

[5] I. Düntsch, E. Orłowska. A proof system for contact relation algebras, *J. Philosophical Logic*, 29:241-262, 2000.

[6] A. Formisano, E. G. Omodeo, and A. Policriti. Three-variable statements of set-pairing. *Theoretical Computer Science* (special issue in honor of Denis Richard, still to appear), 2004.

[7] A. Formisano, E. G. Omodeo, and M. Simeoni. A graphical approach to relational reasoning. In W. Kahl, D. L. Parnas, and G. Schmidt Eds. *Electronic Notes in Theoretical Computer Science*, **44**(3), Elsevier Science, 2003.

[8] A. Formisano, E. G. Omodeo, and M. Temperini. Layered map reasoning: An experimental approach put to trial on sets. In A. Dovier and M. C. Meo and A. Omicini Eds. *Electronic Notes in Theoretical Computer Science*, **48**, Elsevier Science, 2001.

[9] A. Formisano, E. G. Omodeo, and M. Temperini. Instructing equational set-reasoning with Otter. In R. Goré and A. Leitsch and T. Nipkow Eds., *Automated Reasoning. Proc. of First International Joint Conference on Automated Reasoning* (IJCAR'01–(CADE+FTP+TABLEAUX), Siena), Lecture Notes in Computer Science **2083**, pp. 152-167, Springer, 2001.

[10] A. Formisano and A. Policriti. T-resolution: refinements and model elimination. *J. of Automated Reasoning*, **22**(4):433–483, 1999.

[11] A. Formisano and M. Simeoni. An AGG application supporting visual reasoning. In L. Baresi and M. Pezzè Eds. *Electronic Notes in Theoretical Computer Science*, **50**(3), Elsevier Science, 2001.

[12] P. R. Halmos. *Algebraic Logic.* Chelsea, New York, 1962.

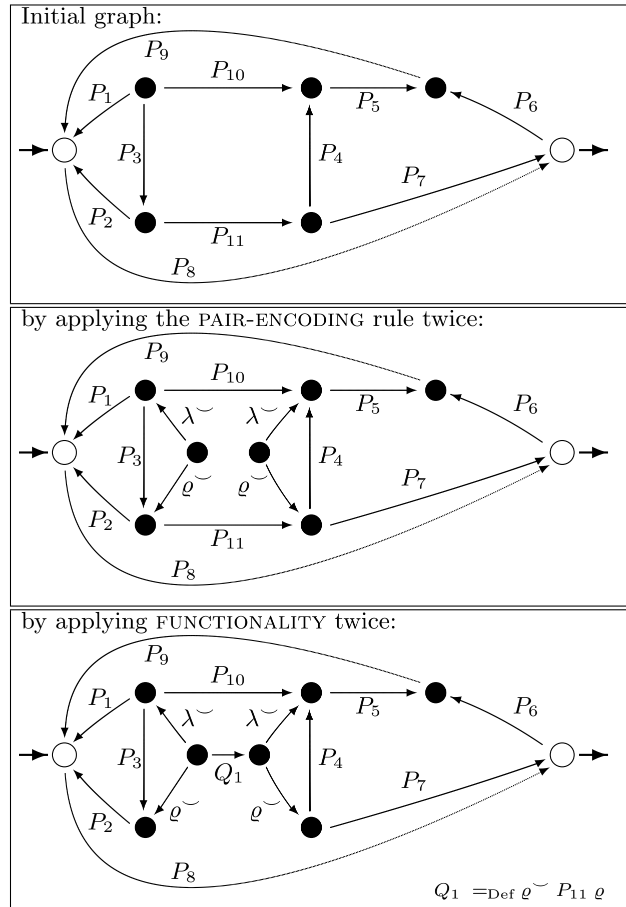


Fig. 6. Rehabilitation of an irreducible graph (continued in Figure 7)

[13] L. Henkin. Completeness in the theory of types. In J. Hintikka ed., *The philosophy of mathematics*, Oxford readings in Philosophy, Oxford University Press, 1969, pp.51–63 (1950).

[14] E. Omodeo, E. S. Orłowska, and A. Policriti. Rasiowa-Sikorski style relational elementary set theory. In R. Berghammer and B. Moeller Eds., *Proc. 7th International Seminar on Relational Methods in Computer Science* (RelMiCS'03, May 12–17, 2003, Malente, Germany).

[15] E. S. Orłowska. Relational semantics for nonclassical logics: Formulas are relations, In J. Wolenski ed., *Philosophical Logic in Poland*, pp. 167-186, 1994.

[16] G. Schmidt and T. Ströhlein. *Relations and Graphs –Discrete Mathematics for Computer Scientists.* EATCS-Monographs on Theoretical Computer Science, Springer, 1993.

[17] A. Tarski and S. Givant. *A formalization of set theory without variables*, vol. **41** of *Colloquium Publications.* American Mathematical Society, 1987.

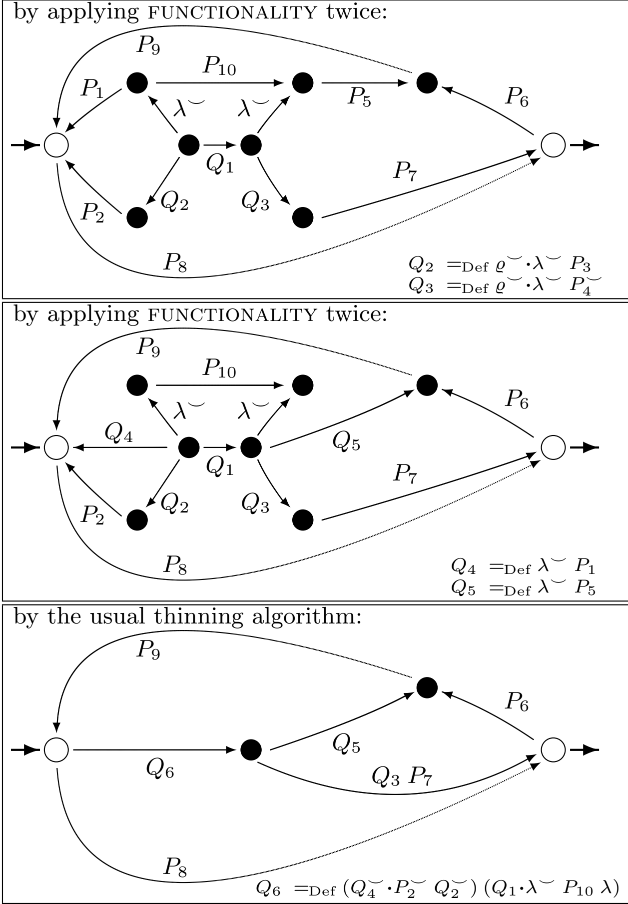


Fig. 7. Rehabilitation of an irreducible graph

[18] L. A. Wallen. *Automated proof-search in non-classical logics – Efficient matrix proof methods for modal and intuitionistic logics*, The MIT Press, Cambridge, MA, and London, 1990.

Constraint Based Protein Structure Prediction Exploiting Secondary Structure Information

Alessandro Dal Palù[2], Sebastian Will[1], Rolf Backofen[1], and Agostino Dovier[2]

[1] Jena Center of Bioinformatics, Institute of Computer Science, Friedrich-Schiller-University, Jena. Ernst-Abbe-Platz 2, 07743 Jena (Germany).

[2] Department of Mathematics and Computer Science, University of Udine. Via delle Scienze 206, 33100 Udine (Italy).

Abstract. The protein structure prediction problem is one of the most studied problems in Computational Biology. It can be reasonably abstracted as a minimization problem. The function to be minimized depends on the distances between the various amino-acids composing the protein and on their types. Even with strong approximations, the problem is shown to be computationally intractable. However, the solution of the problem for an arbitrary input size is not needed. Solutions for proteins of length 100–200 would give a strong contribution to Biotechnology. In this paper, we tackle the problem with constraint-based methods, using additional constraints and heuristics coming from the secondary structure of a protein that can be quickly predicted with acceptable approximation. Our prototypic implementation is written using constraints over finite domains in the Mozart programming system. It improves over any previous constraint-based approach and shows the power and flexibility of the method. Especially, it is well suited for further extensions.

1 Introduction

A protein is identified by a finite list of amino-acids, which we can represent as symbols of an alphabet of 20 elements. The *protein structure prediction (PSP)* problem is the problem of predicting the 3D structure of the protein (its *native conformation*) when the list of amino-acids is known. Native conformation determines the Biological function of the protein. It is accepted that the native conformation is the state of minimum free energy. Up to now, though, a definitive energy model has not been yet devised. The energy of a conformation depends partially on the *distances* between all pairs of amino-acids and on their *type*. Thus, the PSP problem can be simplified to that of minimizing a suitable energy function generated by the protein 3D conformations. Although the problem, even with some simplifications, is NP-complete [9], it deserves to be attacked, because a solution for 'small' proteins (100–200 amino-acids) would be anyhow extremely important in Biology and Biotechnology. The problem is approached in several ways (see [7, 14] for a review). Prediction methods make use of statistical information available from more than 24,000 structures deposited in the Protein Data Bank (PDB) [5]. The correct fold for a new sequence can be obtained when *homology* (sequence similarity) is detected with a sequence for

which the structure is available. Another approach tries to superimpose (*thread*) a chain on known structures and evaluates the plausibility of the fold.

Ab-initio methods, instead, try to find the native conformation without direct reference to a structural model. In this context, a constraint-based encoding of the problem is extremely natural. A first abstraction is the choice of a spatial model for the admissible positions of the various amino-acids. *Lattice models* are used to formalize the PSP problem as a minimization problem on finite domains. It has been shown [4, 3, 12] that the Face-Centered-Cubic lattice (FCC) provides the best lattice approximation of proteins. Moreover, it has been shown that the residue neighbors in real proteins are clustered in a rather dense way, occupying positions closely approximating those of a distorted FCC packing. It is believed that this packing is a direct manifestation of the hydrophobic effect. Recall that FCC is the closest packing of spheres, which was proven only recently [8].

The FCC lattice is exploited in [1, 16] for solving proteins of length up to 160 with the further abstraction of splitting the amino-acids in two families (H and P). However, the HP abstraction does not ensure that the result is the native conformation; in particular, the local sub-conformations of the form of α-helices or β-strands (cf. Sect. 2) are often lost. These structures are often generated early in the real folding process and, moreover, the structures can be predicted with good approximation [13]. These two observations suggest to include secondary structures in the constraint-based definition of the (complete, non-HP) problem. This is done in [10] using $CLP(\mathcal{FD})$ constraints in SICStus Prolog; proteins of length 50–60 can be predicted.

In this paper we extend the results of [10] in three ways. First, a precise mathematical formalization of secondary structure is given and results are obtained, which allow for easy breaking of symmetries, cheap computation of energy, and an effective, but computationally inexpensive, search strategy during the constraint search. Then we split the energy function into 4 families and we observe that the energy contribution of one of them is sensibly greater than the others: we use this information for developing heuristics for the solution's search process. Finally, we take advantages from using the constraint propagation of Oz 3.0 language and in particular we implement specially tailored propagators that allow to solve the problem efficiently. The preliminary results obtained show a huge speed-up w.r.t. the results of [10].

The paper is organized as follows: in Section 2 we provide some biological background needed through the paper. In Section 3 we describe the spatial and protein models. In Section 4 we define the problem and introduce our solving strategy. In Sections 5 and 6 we provide technical details to handle secondary structure elements. In Section 7 we describe the constraint framework. In Section 8 we present some preliminary results and we conclude with Section 9.

2 Biological Background

The *Primary* structure of a protein is a finite sequence of linked units (or *residues*), that define uniquely the molecule. Each residue is an amino-acid,

denoted by one of the 20 elements in the alphabet set Σ. A *protein (primary) sequence* is thus a string $s \in \Sigma^*$.

Native conformations are largely built from *Secondary Structure elements (*SSE*s)*, which are local motifs consisting of short, consecutive parts of the amino acid sequence having a very regular conformation. Some of them are α-helices, β-sheets, and $\beta\alpha\beta$ turns. In this paper we model the first two motifs: α-helices are constituted by 5 to 40 residues arranged in a regular right-handed helix with 3.6 residues per turn. This local structure is stabilized by local interactions and can be thought as a rigid cylinder. β-sheets are constituted by extended strands of 5 to 10 residues. Each strand is made of contiguous residues, but strands participating in the same sheet are not necessarily contiguous in sequence. There are algorithms based on neural networks that can predict with high accuracy (75% [7]) the secondary structure of a protein. Formally, a *secondary structure for a protein sequence* $s = s_1 \dots s_n$ is a list sse of k triples $SSE_i = (t_i, b_i, e_i)$, where for $0 \leq i < k$, $t_i \in \{\boldsymbol{\alpha}, \boldsymbol{\beta}\}$, $0 \leq b_i < e_i < n$, and for $0 \leq i < k-1$, $e_i < b_{i+1}$. Later, we will use the notation $|s| = n$, and $|\text{sse}| = k$. Given a secondary structure sse for a protein sequence s, then the *sub-sequence of the i-th* SSE is the string $s_{b_i} \dots s_{e_i}$.

In nature, each protein always reaches a specific 3D conformation, called native conformation or *tertiary structure*. This conformation determines the function of the protein. The *protein structure prediction problem* is the problem of determining the tertiary structure of a protein given its primary structure. It is accepted that the primary structure uniquely determines the tertiary structure. Due to entropic considerations, it is also accepted that the tertiary structure minimizes the global energy of the protein. Though, is not yet uniquely accepted which energy function describes this phenomenon.

3 Formalizing the Models

3.1 The Lattice Model

In this section, we introduce the spatial model and its properties. The protein is represented as a succession of three dimensional points. Each point corresponds to an amino-acid[3]. In our approach we restrict the point's domain to be in the *face-centered-cubic lattice (FCC)*. Fig 1a) shows its unit cell. The FCC is the set of points $\mathsf{FCC} = \{(x, y, z) | x, y, z \in \mathbb{Z}, x + y + z \text{ is even}\}$.

We discuss now some important properties of this lattice. First, note that the FCC has 48 automorphisms, which are represented by orthogonal matrices M, where the column vectors are a permutation of $(\pm 1, 0, 0), (0, \pm 1, 0)$, and $(0, 0, \pm 1)$. As we now will explain, we use only 24 of these automorphisms, due to the chirality of proteins.

Chirality is an important property of protein structures and sub-structures. Two objects are *chiral* if they are identical except for a mirror reflection (different *handedness*). In nature, a sequence of amino-acids, when folded (e.g. right

[3] In particular, we associate each point to the position of the α-carbon lying in the backbone of the amino-acid, which can be assumed to be its steric center.

handed α-helices), has a specific handedness and usually it can not generate the symmetric folding as well. In our model, when we apply transformations, we are interested in preserving the natural handedness. Since reflections invert the handedness, we restrict to automorphisms with an orthogonal matrix with positive determinant. Note that half of the possible automorphisms are reflections, since the determinant associated to the matrix is negative. This leads us to the following definition of the basic transformation that we apply to points:

Definition 1. *A matrix M is called* rotational*, if it is orthogonal,* $\det(M) = 1$ *and each of its elements is in $\mathbb{Z}$. An* isometric mapping *$\langle M, \boldsymbol{t}\rangle$, where $\boldsymbol{t} \in \mathsf{FCC}$ and M is a rotational matrix, is a function from points to points of the form $\langle M, \boldsymbol{t}\rangle : \boldsymbol{p} \to M\boldsymbol{p} + \boldsymbol{t}$. We identify this function with its extension to sets of points. M is called the* transformation matrix *of the isometric mapping $\langle M, \boldsymbol{t}\rangle$ and $\boldsymbol{t}$ its* translation vector*.*

The composition $A_2 \circ A_1$ of two isometric mappings $A_2 = \langle M_2, \boldsymbol{t}_2\rangle$ and $A_1 = \langle M_1, \boldsymbol{t}_1\rangle$, i.e. the application of A_2 after A_1 equals

$$\langle M_2, \boldsymbol{t}_2\rangle \circ \langle M_1, \boldsymbol{t}_1\rangle = \langle M_2 M_1, M_2\boldsymbol{t}_1 + \boldsymbol{t}_2\rangle.$$

Note that $A_2 \circ A_1$ is again an isometric mapping, since M_2M_1 is rotational. Due to this definition, there is also an inverse for an isometric mapping $\langle M, \boldsymbol{t}\rangle$, namely $\langle M, \boldsymbol{t}\rangle^{-1} \triangleq \langle M^{-1}, -M^{-1}\boldsymbol{t}\rangle$. We define the FCC-norm of a point (x, y, z):

$$\|\boldsymbol{p}\|_{\text{fcc}} = \max\left\{|x|, |y|, |z|, \frac{|x| + |y| + |z|}{2}\right\}.$$

A vector $\boldsymbol{v} \in \mathsf{FCC}$ is called a *unit vector* if $\|\boldsymbol{v}\|_{\text{fcc}} = 1$. The FCC-distance of two points $\boldsymbol{p}, \boldsymbol{q} \in \mathsf{FCC}$ is $\|\boldsymbol{p} - \boldsymbol{q}\|_{\text{fcc}}$. Let us observe that in the FCC lattice, each walk from $(0, 0, 0)$ to $(x, y, z) \in \mathsf{FCC}$, needs at least $\|(x, y, z)\|_{\text{fcc}}$ lattice unit vectors. We also use the standard Euclidean norm $\|\langle x, y, z\rangle\|_2 = \sqrt{x^2 + y^2 + z^2}$.

3.2 The Protein Model

We provide here the formal definition of our 3D model and energy function. The tertiary structure of a protein can be modeled as a function $\omega : [0 \,..\, n{-}1] \to \mathsf{FCC}$. The position of the i-th monomer in the protein corresponds to the point $\omega(i)$.

Definition 2. *The function ω is a* folding *for the primary sequence s, iff*
bond-constraint: $\forall i \,.\, 0 \leq i < |s| - 1 : \|\omega(i) - \omega(i+1)\|_{fcc} = 1$*, and*
angle-constraint: $\forall i \,.\, 0 \leq i < |s|{-}2 : \angle(\omega(i), \omega(i{+}1), \omega(i{+}2)) \in \{90°, 120°, 180°\}$.

Note that we allow only angles of 90°, 120°, and 180°, whereas in the FCC lattice, three consecutive monomers can also form angles of 0° and 60°. We exclude the angles less than 90° for sterical reasons, since the first and third amino-acid would be too close. Note that in [10] the 180° angle was excluded as well, since it is unfavorable in real proteins. In our approach we allow this angle, since simpler constraints can be used (especially when linking SSEs to

neighbors). Moreover, when discretizing the protein on FCC lattice, 180° angles can be required to fit more accurately the native state. For example, the modeling of α-helices in the FCC lattice, is not able to represent the typical periodicity of the pattern.

A folding ω *satisfies* a secondary structure $\text{sse} = (t_i, b_i, e_i)_{0 \le i < |\,\text{sse}\,|}$ if and only if for every $i \in [0..|\,\text{sse}\,| - 1]$, the positions $\omega(b_i), \ldots, \omega(e_i)$ approximate a right-handed α-helix on the FCC if $t_i = \boldsymbol{\alpha}$ and approximate a β-strand on the FCC if $t_i = \boldsymbol{\beta}$. We give a formal definition of these approximations in Subsection 5.1.

The modeling of the folding energy is a delicate issue: a large set of phenomena can be included to produce a refined energy function. In this paper we restrict to one kind of interaction between elements, namely the *contact energy* between pairs of amino-acids and we use the matrix developed in [6]. $\text{ppot}(a, b)$ denotes the potential of the amino-acids a and b. The potential is only contributed to the total energy if the two amino-acids are in close contact. If two monomers are too close, they are *clashing*, in this case the total energy is ∞, since for steric reasons two residues repel each other.

Definition 3 (Energy). *For two amino-acids $a, b \in \Sigma$ and two positions $\boldsymbol{p}, \boldsymbol{q} \in$ FCC, we define*

$$\mathrm{E}(a, b, \boldsymbol{p}, \boldsymbol{q}) = \begin{cases} \text{ppot}(a, b) & \|\boldsymbol{q} - \boldsymbol{p}\|_2 = 2 \\ \infty & \|\boldsymbol{q} - \boldsymbol{p}\|_2 < 2 \\ 0 & \textit{otherwise.} \end{cases}$$

The energy of a protein conformation with protein sequence s and folding $\omega : [0 \,..\, |s| - 1] \to \mathsf{FCC}$ *is* $\mathrm{E}^{\mathrm{C}}(s, \omega) = \sum_{0 \le i, j < |s|, i+2 < j} \mathrm{E}(s_i, s_j, \omega(i), \omega(j))$.

Note that due to our definition, every sub-sequence up to 3 consecutive amino-acids in the primary structure, does not contribute to the energy. This is an empirical choice. Basically, if three consecutive amino-acids formed an angle of 90° the first and third amino-acid would form a pair contributing to the energy, whereas if they formed an angle of 120° there would be no contribution. The energy contribution of an amino-acid pair is negative on average and thus, angles of 90° would be always favoured. This is true in the lattice, but not in nature, and thus we avoid this a-priori preference by removing the energy contribution of the pair of amino-acids s_i and s_{i+2}.

We report here a part of the potential table of [6], also available at `http://www.dimi.uniud.it/dovier/PF/`.

	CYS	MET	PHE	ILE	LEU	VAL	TRP	TYR	ALA	GLY
CYS	-3.477	-2.240	-2.424	-2.410	-2.343	-2.258	-2.080	-1.892	-1.700	-1.101
MET	-2.240	-1.901	-2.304	-2.286	-2.208	-2.079	-2.090	-1.834	-1.517	-0.897
PHE	-2.424	-2.304	-2.467	-2.530	-2.491	-2.391	-2.286	-1.963	-1.750	-1.034
...	...									

4 Problem Definition and Solving Strategy

We define the protein structure prediction problem when the secondary structure information is taken into account. Given a sequence s and a secondary structure

sse, the problem is to find the folding ω that satisfies sse and has minimal energy $\mathrm{E}^{\mathrm{C}}(s,\omega)$. The energy function can be partitioned as a sum of four energy terms, namely the energy contribution by pairs of amino-acids, where

1. both are in the same SSE,
2. both are in SSEs, but not in the same,
3. one is in a SSE and the other is not,
4. both are not in SSEs.

Formally, these contributions, whose sum is $\mathrm{E}^{\mathrm{C}}(s,\omega)$ are defined as

1. $\mathrm{E}^{\mathrm{s}}(s,\mathrm{sse},\omega) = \sum_{0\leq r<|\,\mathrm{sse}\,|} \sum_{b_r \leq i+2<j\leq e_r} \mathrm{E}(s_i, s_j, \omega(i), \omega(j))$
2. $\mathrm{E}^{\mathrm{ss}}(s,\mathrm{sse},\omega) = \sum_{0\leq r<r'<|\,\mathrm{sse}\,|} \sum_{b_r\leq i\leq e_r} \sum_{b'_r\leq j\leq e'_r, i+2<j} \mathrm{E}(s_i, s_j, \omega(i), \omega(j))$
3. $\mathrm{E}^{\mathrm{sn}}(s,\mathrm{sse},\omega) = \sum_{i\in \mathrm{D}} \sum_{0\leq j<|s|, j\notin \mathrm{D}, i+2<j} \mathrm{E}(s_i, s_j, \omega(i), \omega(j))$
4. $\mathrm{E}^{\mathrm{nn}}(s,\mathrm{sse},\omega) = \sum_{0\leq i<j<|s|, i,j\notin \mathrm{D}, i+2<j} \mathrm{E}(s_i, s_j, \omega(i), \omega(j))$,

where D is the set of positions in SSE, i.e. $\bigcup_{0\leq r<|\,\mathrm{sse}\,|}[b_r \mathinner{..} e_r]$. The first term $\mathrm{E}^{\mathrm{s}}(s,\mathrm{sse},\omega)$ is constant for each folding ω that satisfies a given secondary structure sse. Thus, optimizing $\mathrm{E}^{\mathrm{C}}(s,\omega)$ it is equivalent to optimize $\mathrm{E}^{\mathrm{ss}}(s,\mathrm{sse},\omega) + \mathrm{E}^{\mathrm{sn}}(s,\mathrm{sse},\omega) + \mathrm{E}^{\mathrm{nn}}(s,\mathrm{sse},\omega)$.

From a set of 500 selected PDB-proteins, we estimated the average contributions of the last three terms to their sum in the native state, which is given by the following distributions: E^{ss} 49% , E^{sn} 36%, and E^{nn} 15%. Note that half of the energy is contributed by interaction between SSEs. This suggests a heuristic to solve the structure prediction problem. Thus, first we place the SSEs optimally according to E^{ss}. Then, for fixed SSEs, we place the remaining amino-acids while optimizing $\mathrm{E}^{\mathrm{C}}(s,\omega)$.

During the optimization of E^{ss}, the SSEs are placed in the FCC lattice. The elements are treated as rigid blocks that can be shifted and oriented in the lattice. It is fundamental to be independent from global rigid transformations, which give the same isomorphic results. In the next section, we introduce the notion of relative position between two SSEs. This concept is the base to abstract from the symmetries of the problem. Note that all energy terms are based on the notion of distance between amino-acids, thus they are invariant under rigid transformations. The relative position description, thus, is suitable to be associated to a specific energy contribution provided by the represented class.

5 Absolute and Relative Positions

In this section, we discuss a formalism to describe the SSEs and their placement in the lattice. We specify an object (later denoted by *template*) by a list of FCC points. The object is placed in the space by transforming these points by means of rotating and translating. This transformation defines the absolute position of the placed object (later denoted by *instance*). We also introduce the relative position of two SSEs, which is the transformation required to map the absolute position of the first element into the one of the second. Fig. 1b) provides an illustration of the concepts that are introduced in this section.

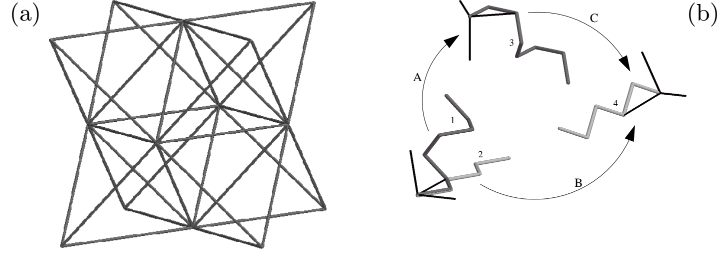


Fig. 1. a) Unit Cell of the face-centered cubic lattice (FCC). There is one point in each corner of a cube and one point in the center of each face. We show the connections by unit vectors. b) Idea of absolute and relative positions. The figure shows instances a helix template of length 8 and type 0 and a sheet template of length 6. If the instances 1 and 2 are in absolute position id, then A (resp. B) is the absolute position of the instance 3 (resp. 4). $C = \rho(A, B)$ is the relative position between the instances (from A to B). We show local coordinate systems for each instance to illustrate the transformation.

5.1 Templates

A *template* is a function $T : [0 \mathinner{..} \ell - 1] \to \mathsf{FCC}$, where $\ell \in \mathbb{N}$ is its *length*. We introduce two classes of templates, namely α-helices and β-sheets. In these cases, a template is the geometric description of a helix (sheet) starting from $(0, 0, 0)$.

Definition 4 (Templates for SSEs). *Let a function* $h : \mathbb{N} \to \mathsf{FCC}$ *be given by* $h(4i+0) = (0,0,0) + i\,\boldsymbol{d}_h$, $h(4i+1) = (1,0,1) + i\,\boldsymbol{d}_h$, $h(4i+2) = (2,0,0) + i\,\boldsymbol{d}_h$, *and* $h(4i+3) = (2,1,-1) + i\,\boldsymbol{d}_h$ *for* $i \in \mathbb{N}$, *where* $\boldsymbol{d}_h = (2,2,0)$. *Then, the* helix template of length n and type τ, *where* $n \in \mathbb{N}$ *and* $\tau \in \{0, 1\}$, *is the function* $\mathrm{helix}_n^\tau : [0 \mathinner{..} n-1] \to \mathsf{FCC}$, $\mathrm{helix}_n^\tau(i) = h(i - \tau) \quad (i \in [0 \mathinner{..} n-1])$.

Furthermore, let a function $\mathrm{sheet}_n : [0 \mathinner{..} n-1] \to \mathsf{FCC}$, *be given by* $s(2i+0) = (0,0,0) + i\,\boldsymbol{d}_s$ *and* $s(2i+1) = (1,1,0) + i\,\boldsymbol{d}_s$ *for* $i \in \mathbb{N}$, *where* $\boldsymbol{d}_s = (2,0,0)$.

Note that the helix templates of type 0 (resp. 1) describe helices, where the first three points form an angle of 90° (resp. 120°).

5.2 Position of an Instance

Definition 5 (Absolute Position, Instance). *An* absolute position *is an isometric mapping. An* instance I with absolute position $\langle M, \boldsymbol{t} \rangle$ of a template T *is a pair* $I = \langle M, \boldsymbol{t} \rangle \Diamond T$. *The* instance function of I, *denoted by* I^{fun}, *is* $\langle M, \boldsymbol{t} \rangle \circ T$.

For $i \in \mathbb{Z}$, *we use the short notations* $I(i)$ *for* $I^{fun}(i)$ *and* $\mathrm{dom}(I)$ *for* $\mathrm{dom}(I^{fun})$. *The* image of an instance I *is* $\mathrm{img}(I) = \{I(i) | i \in \mathrm{dom}(I)\}$.

The instance function of the instance $\langle \mathrm{id}, (0,0,0) \rangle \Diamond T$ is $\langle \mathrm{id}, (0,0,0) \rangle \circ T = T$.

Definition 6 (Relative Position). *For instances* I_i *with absolute positions* A_i $(i = 1, 2)$, *the* relative position $\rho(A_1, A_2)$ from A_1 to A_2 *is the isometric mapping, where* $\rho(A_1, A_2) = A_1^{-1} \circ A_2$. *Then,* $\rho(A_1, A_2)$ *is also called the* relative position of the instances I_1 and I_2.

The relative position between two instances $I_i = A_i \Diamond T_i$ $(i = 1, 2)$ can be used to obtain the second instance from the first instance as $I_2 = A_1 \circ \rho(A_1, A_2) \Diamond T_2$, since by definition $A_2 = A_1 \circ \rho(A_1, A_2)$.

The following proposition claims that the relative position of two instances is invariant under global transformations.

Proposition 1. *Given the instances $I_i = A_i \Diamond T_i, I'_i = A'_i \Diamond T'_i$ (i=1,2), the relative positions $R = \rho(A_1, A_2), R' = \rho(A'_1, A'_2)$ and the isometric mappings C_i, such that $A'_i = C_i \circ A_i$, then $R = R'$ if and only if $C_1 = C_2$.*

Note that by Proposition 1, $\rho(M \circ A_1, M \circ A_2) = \rho(A_1, A_2)$ for isometric mappings M, A_1, and A_2. Especially, for every relative position $\rho(A_1, A_2)$, there is an identical relative position of the form $\rho(\mathrm{id}, B) = B$ for some isometric mapping B, namely $B = A_1^{-1} \circ A_2$.

6 Energy contribution of instance pairs

In the first phase of the algorithm, we pre-compute the energy contribution of a pair of instances in every relevant placement. Therefore, we enumerate the set of relative positions between the instances of two templates, where the two instances interact sterically. Only for those relative positions, the energy contribution is not equal to zero.

Definition 7 (Interaction). *Two instances I_1 and I_2* interact*, if and only if there exist $i_1 \in \mathrm{dom}(I_1), i_2 \in \mathrm{dom}(I_2)$, such that $\|I_2(i_2) - I_1(i_1)\|_2 \leq 2$. We define the* interaction set of templates T_1 and T_2 *as* $\mathrm{InteractionSet}(T_1, T_2) =$

$$\{R = \rho(\mathrm{id}, R) \mid I_1 = \mathrm{id} \Diamond T_1, I_2 = R \Diamond T_2, I_1 \text{ and } I_2 \text{ interact}\}.$$

Note that in the definition, we have in mind to fix the first instance (here to id) and move the second instance to every interacting position. Nevertheless, due to Proposition 1, the interaction set contains all relative positions between arbitrary instances that are interacting. Note that the interaction set is finite, since the instances are finite.

If we define neighVecs as a tuple of the 19 vectors $\boldsymbol{p} \in \mathsf{FCC}$ with $\|\boldsymbol{p}\|_2 \leq 2$ in arbitrarily fixed order, then for a template T, we define an *extended template* T^{ext} by $T^{\mathrm{ext}}(19 \cdot i + j) = T(i) + \mathrm{neighVecs}_j$, for $i \in \mathrm{dom}(T)$ and $0 \leq j < 19$.

We say that two instances *intersect* if and only if their images have a non-empty intersection.

Proposition 2. *Two instances $I_1 = A_1 \Diamond T_1$ and $I_2 = A_2 \Diamond T_2$ interact if and only if $I_1^{ext} = A_1 \Diamond T_1^{ext}$ and I_2 intersect.*

Definition 8 (Energy Contribution). *Two instances of templates T_i with sequence s_i $(i = 1, 2)$ with relative position R give an energy contribution of*

$$\mathrm{E}^{\mathrm{T}}(s_1, s_2, T_1, T_2, R) = \sum_{0 \leq j < |s_1|, 0 \leq k < |s_2|} \mathrm{E}(s_{1j}, s_{2k}, \mathrm{id} \Diamond T_1(j + b_1), R \Diamond T_2(k + b_2)), \quad (1)$$

where $b_1 = \min(\mathrm{dom}(T_1))$ and $b_2 = \min(\mathrm{dom}(T_2))$.

Recall that for the templates T_1 and T_2, there are only finitely many relative positions R, such that $\mathrm{id}\,\Diamond T_1$ and $R\Diamond T_2$ interact, i.e. $\mathrm{InteractionSet}(T_1, T_2)$ is finite. By merging the two definitions of interaction and the energy contribution of a pair of amino-acids, if $\mathrm{E}^{\mathrm{T}}(s_1, s_2, T_1, T_2, R)$ is different from 0, then $\mathrm{id}\,\Diamond T_1$ and $R\Diamond T_2$ interact. Note that from the interaction of $\mathrm{id}\,\Diamond T_1$ and $R\Diamond T_2$, we can not conclude that $\mathrm{E}^{\mathrm{T}}(s_1, s_2, T_1, T_2, R) \neq 0$, since depending on the table of pairwise potentials certain interaction patterns of the two instances could sum up to 0.

Due to this, in order to completely give the energy for every relative position R it suffices to consider all $R \in \mathrm{InteractionSet}(T_1, T_2)$. Now, the problem discussed in this subsection reduces to generate $\mathrm{InteractionSet}(T_1, T_2)$. Instead of enumerating the relative positions, where instances of T_1 and T_2 interact, we equivalently enumerate the relative positions, where instances of T_1^{ext} and T_2 intersect. Due to the following proposition, we can completely enumerate the set $\mathrm{InteractionSet}(T_1, T_2)$.

Proposition 3. *For templates T_1 and T_2,*

$$\mathrm{InteractionSet}(T_1, T_2) = \biguplus_{M \text{ rot. matrix}} \{\langle M, \boldsymbol{t}\rangle \mid \exists j_1, j_2 : \mathrm{id}\,\Diamond T_1^{ext}(j_1) = \langle M, \boldsymbol{t}\rangle \Diamond T_2(j_2)\}.$$

Proof. Obviously, one can partition any interaction set by the rotation matrices of the relative positions. Then, the proposition is a consequence of Proposition 2.

For the inclusion $\subseteq$, let $R = \langle M, \boldsymbol{t}\rangle$ in $\mathrm{InteractionSet}(T_1, T_2)$. Then, the instances $\mathrm{id}\,\Diamond T_1$ and $R\Diamond T_2$ interact, i.e. the instances $\mathrm{id}\,\Diamond T_1^{\mathrm{ext}}$ and $R\Diamond T_2$ intersect.

For the inclusion $\supseteq$, for any $R = \langle M, \boldsymbol{t}\rangle$ in one of the subsets $\{\langle M, \boldsymbol{t}\rangle \mid \exists j_1, j_2 : \mathrm{id}\,\Diamond T_1^{\mathrm{ext}}(j_1) = \langle M, \boldsymbol{t}\rangle \Diamond T_2(j_2)\}$, the instances $\mathrm{id}\,\Diamond T_1^{\mathrm{ext}}$ and $R\Diamond T_2$ intersect. Thus, $\mathrm{id}\,\Diamond T_1$ and $R\Diamond T_2$ interact. □

The proposition suggests the following algorithm. For every rotation matrix M, for every pair of indices $j_1 \in \mathrm{dom}(T_1)$ and $j_2 \in \mathrm{dom}(T_2)$, collect the unique translation vectors $\boldsymbol{t}$, where $\mathrm{id}\,\Diamond T_1^{\mathrm{ext}}(j_1) = \langle M, \boldsymbol{t}\rangle \Diamond T_2(j_2)$.

Note that there is indeed a unique $\boldsymbol{t}$ for every M, j_1 and j_2, which is calculated in constant time. Also note that whereas this algorithm can enumerate a vector $\boldsymbol{t}$ more than once, the algorithm still calculates only a limited number of $24 \times |\,\mathrm{img}(T_1^{\mathrm{ext}})| \times |\,\mathrm{img}(T_2)|$ many vectors $\boldsymbol{t}$.

7 Constraint Model

Recall, that we discuss the problem of finding the folding ω for a given protein sequence s and a given secondary structure $\mathrm{sse} = (t_i, b_i, e_i)_{0 \leq i < |\,\mathrm{sse}\,|}$, that satisfies sse and has minimal energy $\mathrm{E}^{\mathrm{C}}(s, \omega)$. We already suggested a heuristic for the minimization, which consists of two separate phases and uses the outcome of the already described computation of energy contributions of instances.

In a first branch-and-bound search, we place the secondary-structure elements (SSEs) while optimizing the energy contribution E^{ss}, i.e. we search for a folding ω that satisfies sse and has minimal energy $\mathrm{E}^{\mathrm{ss}}(s, \mathrm{sse}, \omega)$ and a finite

total energy $E^C(s,\omega)$. For finite energy $E^C(s,\omega)$, there must not be clashes in the tertiary-structure. When we place only the SSEs, we can not check the non-clashing of the remaining amino-acids efficiently by consistency methods. Hence, we search for one consistent placing of the remaining amino-acids each time we find a new placement of the SSEs. Then, a second branch-and-bound search computes a placement of SSEs and remaining amino-acids that optimizes the total energy $E^C(s,\omega)$ and places the SSEs nearly optimally w.r.t. E^{ss}.

All the constraints and the enumeration strategy are common for the two phases. Due to this, we are able to give only a common description as well as to implement only a single solver in our implementation language *Oz 3.0* [15].

The constraint model as well as our enumeration strategy divides into two parts. The first part deals with the placement of the SSEs, whereas the second part handles the placement of the remaining amino-acids. The focus of our work is on the first part and only this part shall be described in more detail.

In the first part, we use the pre-computed energy contributions of instance pairs. Therefore, we start with relating our placement problem to the notion of templates and instances.

We define templates for every SSE in sse. For each SSE k, we introduce an absolute position A_k and a type $\tau_k \in \{0,1,2\}$, where $0 \leq k < |\,\mathrm{sse}\,|$. Such a type τ_k combines the distinction between α-helix and β-sheet with the types of helix-templates (cf. Def 4). For a helix, this type gives just the type 0 or 1 of the corresponding helix-template and for sheets this type is always 2. For $0 \leq k < |\,\mathrm{sse}\,|$, where $t_k = \boldsymbol{\alpha}$, we define $T_k^\tau = \mathrm{helix}^\tau_{e_k-b_k+1}$ $(\tau = 0,1)$ and where $t_k = \boldsymbol{\beta}$, we define $T_k^2 = \mathrm{sheet}_{e_k-b_k+1}$. Let s_k denote the sub-sequence of the k-th SSE. We define I_k^τ as $A_k \Diamond T_k^\tau$ for absolute positions A_k. Now, the tertiary-structure ω is related to the instances, by $\omega(i) = A_k \Diamond T_k^{\tau_k}(i - b_k)$ for $i \in [b_k \mathinner{..} e_k]$.

Then, for given template definitions, the positions of amino-acids in SSEs are completely specified by $(A_k)_{0 \leq k < |\,\mathrm{sse}\,|}$ and $(\tau_k)_{0 \leq k < |\,\mathrm{sse}\,|}$. Due to this, we can equivalently investigate the problem in terms of instances. The relation between the instances determines the energy term E^{ss}. In the same time the relations are constrained by the properties of a folding.

We choose to enumerate the relative positions of the elements instead their absolute positions.[4] Besides breaking of symmetries[5], there are several advantages in enumerating relative positions instead absolute positions. Most notably, there is a direct correspondence between the relative position and the energy contribution of a pair of SSEs. This immediate relation is used to dynamically guide the search, i.e. we enumerate highly contributing relations between SSEs first. Furthermore, enumerating relative positions is more general than enumerating absolute positions. Imagine, that we enumerate absolute positions. Then, after w.l.o.g. fixing the absolute position of the first instance to $\langle \mathrm{id}, \mathbf{0} \rangle$, enumerating the absolute positions of the remaining elements is equivalent to enumerating

[4] Notably, the energy contribution E^{ss} is determined only from the relative positions.

[5] Note that by using relative positions instead absolute positions, we break all geometrical symmetries in the problem for free. In general, breaking of symmetries in constraint modeling is a non-trivial task and a broadly discussed topic (e.g. see [2]).

their relative positions to the first element. In contrast, our more flexible enumeration strategy can dynamically decide to enumerate relations earlier that contribute stronger to the total energy than any relation to the first element.

As described in the previous section, we are able to calculate the energy contribution of two instances in every relative position and in particular enumerate the finite list of relative positions, where this energy contribution differs from zero. For every pair of SSEs in sse, we generate such a list of relative positions and corresponding energy contributions. Since for α-helices there are two types of helix-templates, we also include the information on the type in the list for each pair of SSEs. Here, it is convenient to partition the list into two tables: NTab_{ij} and CTab_{ij}. In the former table, we include every relative position that produces a finite, non-zero energy (i.e. there is an interaction but no clash for this relative position), and in the latter we collect all relative positions, where the elements i and j clash (infinite energy). In preparation of our enumeration strategy, the tables NTab_{ij} are ordered by increasing energy-values. For being uniform, we generate for every pair of secondary-structure elements i and j, the tables NTab_{ij} and CTab_{ij}, which consist of all records $(\sigma_i, \sigma_j, R, E)$, where σ_i (resp. σ_j) is a possible value for the type τ_i (resp. τ_j) and $R \in \mathrm{InteractionSet}(T_i^{\sigma_i}, T_j^{\sigma_j})$. Then, E is the corresponding energy contribution $\mathrm{E}^{\mathrm{T}}(s_i, s_j, T_i^{\sigma_i}, T_j^{\sigma_j}, R)$. [6]

7.1 Variables and Constraints

First, for $0 \leq i < j < |\,\mathrm{sse}\,|$ we introduce finite-domain variables $\mathtt{X}_{ij}$, where the domain of the variable $\mathtt{X}_{ij}$ is $[0\,..\,|\,\mathrm{NTab}_{ij}\,| - 1] \uplus \{\Delta\}$. The value of $\mathtt{X}_{ij}$ is either an index in the table NTab_{ij} or Δ. In the first case, the relation (i.e. relative position R_{ij} *and* helix-types) between the SSEs i and j is specified by the $\mathtt{X}_{ij}$-th entry in the table NTab_{ij}. The case $\mathtt{X}_{ij} = \Delta$ represents those relative positions that provide no energy contribution, but cause a distance of elements i and j that still allows to connect the elements in a folding.

Since the variables $\mathtt{X}_{ij}$ completely specify the relative positions only for $\mathtt{X}_{ij} \neq \Delta$, we introduce an explicit representation of relative positions for ensuring consistency. The relative position between the elements i and j is given by variables $\mathtt{R}_{ij}^M$, which encodes for the transformation matrix of R_{ij}, and $\mathtt{R}_{ij}^x$, $\mathtt{R}_{ij}^y$, and $\mathtt{R}_{ij}^z$, which represent the coordinates of the translation vector of R_{ij}. The domain of $\mathtt{R}_{ij}^M$ is finite, since there are only 24 rotational matrices. Also the variables $\mathtt{R}_{ij}^x$, $\mathtt{R}_{ij}^y$, and $\mathtt{R}_{ij}^z$ have finite-domains, since the number of translations is limited due to the bond-constraint connecting the amino-acids between elements i and j. Since the tuple $(\mathtt{R}_{ij}^M, \mathtt{R}_{ij}^x, \mathtt{R}_{ij}^y, \mathtt{R}_{ij}^z)$ represents one relative position, we address this tuple as the variable $\mathtt{R}_{ij}$. For $0 \leq i < |\,\mathrm{sse}\,|$, we add variables $\mathtt{Type}_i \in \{0, 1, 2\}$, which correspond to the types τ_i of the SSE*s*. If $t_i = \boldsymbol{\alpha}$, the value of $\mathtt{Type}_i$ gives the type of the helix 0 or 1. For $t_i = \boldsymbol{\beta}$, we set $\mathtt{Type}_i = 2$. The variables $\mathtt{R}_{ij}$, $\mathtt{Type}_i$ and $\mathtt{Type}_j$ are related to the variable $\mathtt{X}_{ij}$ via

[6] Note that in these tables we represent the rotational matrices of the relative positions by unique indices in the interval $[0\,..\,23]$. This representation is also used to model domains of matrices with integer finite domain variables.

the table NTab_{ij}. This relation is handled by a native propagator[7], which does only cheap propagation, if sufficiently many variables are ground. Furthermore, it checks the validity of the encoded isometric mapping (w.r.t. clashes and the bond-constraint). Also the relation between the relative position variables $\mathtt{R}_{ij}$ that corresponds to the transitivity $R_{ij} \circ R_{jk} = R_{ik}$ is only propagated in such simple cases.

Finally, the energy contribution E^{ss} is computed in a variable $\mathtt{Energy}^{\mathtt{ss}}$ as the sum of variables $\mathtt{Energy}_{ij}$. These variables denote the energy that is contributed by the elements i and j in the relation that is specified by $\mathtt{X}_{ij}$.

The essential propagation in our approach is done by a propagator for the transitivity constraint $R_{ij} \circ R_{jk} = R_{ik}$ on the variables $\mathtt{X}_{ij}$. This propagator has to relate the indices in the domains of the variables $\mathtt{X}_{ij}$ to the corresponding relations. Moreover, it has to handle Δ-values in the domains correctly. For this reason, this propagator distinguishes several cases. *Case 1)* At least two of the variables are determined to Δ. In this case no further propagation can be applied. *Case 2)* The domains of all three variables contain Δ. Then, at this stage we can not derive any information. *Case 3)* The domains of exactly two variables contain Δ. Here, the domain without Δ is used to prune the other two domains. W.l.o.g. let us assume that $\mathtt{X}_{jk}$ and $\mathtt{X}_{ik}$ contain Δ. We now describe how to prune the domain of $\mathtt{X}_{jk}$. For every isometric mapping B indexed by $\mathtt{X}_{jk}$ we check if there exists at least one isometric mapping A indexed by $\mathtt{X}_{ij}$ such that $A \circ B$ is indexed by a value of $\mathtt{X}_{ik}$. If there is no support, the index is removed from the domain of $\mathtt{X}_{jk}$. The analogous procedure is applied to prune $\mathtt{X}_{ik}$. *Case 4)* The domain of at most one variable contains Δ. W.l.o.g. assume that $\mathtt{X}_{ik}$ contains Δ. Then, we collect the composition of every pair from the domains of $\mathtt{X}_{ij}$ and $\mathtt{X}_{jk}$ and intersect the result with the domain of $\mathtt{X}_{ik}$. Moreover, we prune the domains of $\mathtt{X}_{ij}$ and $\mathtt{X}_{jk}$ using the technique of the previous item. Note that during this propagation, we compose only isometric mappings from the tables with consistent types in the same table rows.

To give some implementation details, note that in Oz, every index variable has a range that begins from 1. Moreover, since Oz does not support negative values in finite domains, we shift the domains of variables $\mathtt{R}^x_{ij}, \mathtt{R}^y_{ij}, \mathtt{R}^z_{ij}$ and in all energy variables accordingly. For convenience, we represent the Δ value differently for each i, j with the integer value $|\,\mathrm{NTab}_{ij}\,|$. Due to the complex propagation of the transitivity constraint for the variables $\mathtt{X}_{ij}$, we preferred to implement the propagator in C++, recalling it inside the Mozart code.

7.2 Search Strategy

For placing the SSEs, we enumerate the variables $\mathtt{X}_{ij}$ in a dynamic enumeration order, which combines a first-fail strategy with a preference for variables $\mathtt{X}_{ij}$ that allow a strong contribution to the energy E^{ss}. Here, we find the best contribution that is *allowed by the variable* $\mathtt{X}_{ij}$, if we look up in the table NTab_{ij} using the lowest possible value of $\mathtt{X}_{ij}$ as index. Recall that the tables NTab_{ij} are ordered

[7] In Oz, a native propagator is a propagator implemented in C++. For the purpose of writing special propagators, Oz is extensible via its C++-interface [11].

by increasing energy, in particular for this purpose. As value we will always select the minimal value in the domain, which again corresponds to the optimal energy contribution of the elements i and j.

We will only enumerate the variables $\mathtt{X}_{ij}$ in order to find a good placement of SSEs. Note that since we can assign Δ to variables $\mathtt{X}_{ij}$, not every assignment of these variables completely determines a placement. However, we will only consider placements that are completely determined after the enumeration of the variables $\mathtt{X}_{ij}$. This is justified as a heuristic, since we only search for energetically good placements and such placements will have many strong interactions between the elements. In this situation, there are usually many variables $\mathtt{X}_{ij} \neq \Delta$, which then determine the remaining relative positions.

In practice, we apply also a filtering of the domains of the variables $\mathtt{X}_{ij}$ by their energy. Since the special element Δ also represents the filtered elements, the constraint problem is not changed, except that the energy is calculated less accurately. Filtering these domains has two conflicting impacts. Whereas strong filtering on the domain sizes results in faster enumeration, larger domains achieve a stronger propagation (due to the transitivity constraints). For that reason, we apply two filters. First, we filter the tables NTab to contain only entries with energies in a range of $x\%$ of the optimal energy. The filtered tables constitute the domains of the variables $\mathtt{X}_i j$. The second filtering affects only the enumeration, where we enumerate only values within a range of $y\%$ $(y < x)$ of the optimum. With careful filtering we are able to reduce search times, while preserving a good quality of the results. The strategy is justified, since good placements have usually sufficiently many high scoring pairwise energy contributions.

After the SSEs are placed, we compute absolute positions of their amino-acids by fixing the absolute position of the first element (thereby breaking the symmetries). Then, the positions of the remaining amino-acids are enumerated using a first-fail strategy combined with a preference for good energy contribution. Constraints ensure non-clashing and calculate the remaining energy term $\mathrm{E}^{\mathrm{nn}} + \mathrm{E}^{\mathrm{sn}}$ in a variable $\mathtt{Energy}^{\mathtt{nss}}$.

8 Results

We implemented the described constraint-model in the programming language Oz 3.0 [15] using its most recent implementation Mozart 1.3.0. We extended the language by special constraint propagators written in C++. Also the computation of the energy contributions of instance-pairs is implemented in C++. The implementations are available via `http://www.bio.inf.uni-jena.de`.

For evaluating our result, we ran the prediction for proteins with known structure from the PDB [5]. All predictions are performed on a Pentium 4 at 2.4GHz. Figure 2 shows our prediction for the protein domain "Maternal effect protein Staufen" with PDB-code 1STU in comparison to the known tertiary structure. The protein consists of 68 residues, forming two α-helices and three β-sheets. The computation of the energy contribution of instances was performed in 14 seconds. In the first search, we found a good placement of the SSEs in 3.5

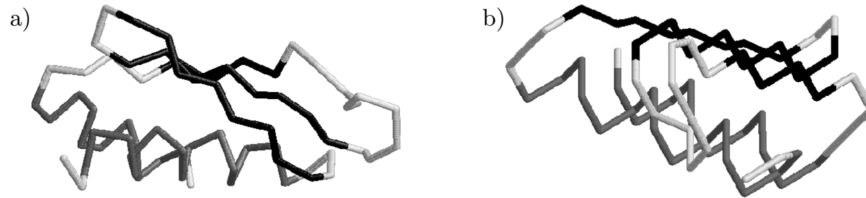


Fig. 2. "Maternal effect protein Staufen". a) backbone of the structure in the PDB (1STU, model 1) b) prediction of our algorithm

minutes, which we could not improve in 7 minutes of search. For the optimal placement of the secondary structure from the first search, we performed a second search for an energetically good tertiary structure placing the remaining amino-acids. The shown solution was found after 9m and could not be improved in 20m. Note that we find structures with only slightly lower energy after 62s of search. We applied the filtering described in Sub-section 7.2 using 30% and 20%.

Furthermore, we compared our approach to the one of [10]. We predicted structures for three proteins from the PDB, which were folded there also. We list PDB-code, number of residues, numbers of SSEs, and run-times for each protein. For our approach, we list the run-times of the three phases separately.

	length	number of SSEs		run-times of phases			run-times of [10]
		α-helices	β-sheets	1	2	3	
1VII	36	3	0	1.6s	3.1s	32s	6m56s
1E0M	37	0	3	0.3s	17s	1m42s	9m45s
2GP8	40	2	0	3.1s	60ms	1.8s	9m0s

Currently, we do not always find good solutions by applying our strategy. Possible reasons and improvements are discussed in the next section.

9 Conclusion and Future Work

We present a novel application of constraint programming to the protein structure prediction problem. The proposed approach combines the use of secondary structure annotation with a strategy that bases tertiary-structure predictions on energetically good placements of SSEs. As we demonstrate using examples from nature, this approach improves in effectivity and application range over recent constraint-based structure prediction algorithms.

However, our main goal in this work is to investigate a basic pattern of a constraint-based protein structure prediction algorithm that is based on secondary-structure information. We plan build on this work in order to improve both, efficiency of the structure prediction and accuracy in modeling proteins.

Regarding the efficiency of structure prediction, please note again, that this work focuses on the placement of SSEs. In consequence, the placement of the remaining amino-acids leaves room for further optimization. Nevertheless, the current strategy turned out to be effective in the discussed application range. A more sophisticated strategy becomes in particular important when investigating improvements in terms of accuracy.

For the aspect of accuracy, we consider it promising to investigate in particular four improvements: refinement of solutions by stochastic optimization, using more complex energy functions (e.g., Lennard-Jones potential), modeling the tertiary-structure off-lattice, and modeling sidechains of amino-acids.

Acknowledgments

The authors thank Federico Fogolari for several useful discussions. A. Dal Palù and A. Dovier are partially supported by MIUR Project *Verifica di sistemi reattivi basata su vincoli (COVER)* and by FSE project *Misura D4*.

References

1. R. Backofen. The protein structure prediction problem: A constraint optimization approach using a new lower bound. *Constraints*, 6(2–3):223–255, 2001.
2. R. Backofen and S. Will. Excluding symmetries in constraint-based search. *Constraints*, 7(3):333–349, 2002.
3. Z. Bagci, R. L. Jernigan, and I. Bahar. Residue coordination in proteins conforms to the closest packing of spheres. *Polymer*, 43:451–459, 2002.
4. Z. Bagci, R. L. Jernigan, and I. Bahar. Residue packing in proteins: Uniform distribution on a coarse-grained scale. *J Chem Phys*, 116:2269–2276, 2002.
5. H. M. Berman, J. Westbrook, Z. Feng, G. Gilliland, T. N. Bhat, H. Weissig, I. N. Shindyalov, and P. E. Bourne. The protein data bank. *Nucleic Acids Research*, 28:235–242, 2000. `http://www.rcsb.org/pdb/`.
6. M. Berrera, H. Molinari, and F. Fogolari. Amino acid empirical contact energy definitions for fold recognition in the space of contact maps. *BMC Bioinformatics*, 4(8), 2003.
7. R. Bonneau and D. Baker. Ab initio protein structure prediction: progress and prospects. *Annu. Rev. Biophys. Biomol. Struct.*, 30:173–89, 2001.
8. B. Cipra. Packing challenge mastered at last. *Science*, 281:1267, 1998.
9. P. Crescenzi, D. Goldman, C. Papadimitrou, A. Piccolboni, and M. Yannakakis. On the complexity of protein folding. In *Proc. of STOC*, pages 597–603, 1998.
10. A. Dal Palù, A. Dovier, and F. Fogolari. Protein folding in $CLP(\mathcal{FD})$ with empirical contact energies. In *Recent Advances in Constraints*, volume 3010 of *Lecture Notes in Computer Science*, pages 250–265. Springer-Verlag, Berlin, 2004.
11. T. Müller and J. Würtz. Interfacing propagators with a concurrent constraint language. In *JICSLP96 Post-conference workshop and Compulog Net Meeting on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, pages 195–206, 1996.
12. B. H. Park and M. Levitt. The complexity and accuracy of discrete state models of protein structure. *Journal of Molecular Biology*, 249(2):493–507, 1995.
13. B. Rost and C. Sander. Prediction of protein secondary structure at better than 70% accuracy. *Journal of Molecular Biology*, 232(2):584–99, 1993.
14. J. Skolnick and A. Kolinski. Computational studies of protein folding. *Computing in Science and Engineering*, 3(5):40–50, 2001.
15. G. Smolka. The Oz programming model. In *Computer Science Today: Recent Trends and Developments*. Springer-Verlag, Berlin, 1995.
16. S. Will. Constraint-based hydrophobic core construction for protein structure prediction in the face-centered-cubic lattice. In *Proceedings of the Pacific Symposium on Biocomputing 2002 (PSB 2002)*.

Using a theorem prover for reasoning on constraint problems

Marco Cadoli and Toni Mancini

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Via Salaria 113, I-00198 Roma, Italy
cadoli|tmancini@dis.uniroma1.it

Abstract. Specifications of constraint problems can be considered logical formulae. As a consequence, it is possible to infer their properties by means of automated reasoning tools. The purpose of this paper is exactly to link two important technologies: automated theorem proving and constraint programming. We report the results on using a theorem prover and a finite model finder for checking existence of symmetries, checking whether a given formula breaks a symmetry, and checking existence of functional dependencies among groups of predicates. As a side-effect, we propose a new domain of application and a brand new set of benchmarks for ATP systems.

1 Introduction

The style used for the specification of a combinatorial problem varies a lot among different languages for constraint programming. In this paper, rather than considering procedural encodings such as those obtained using libraries (in, e.g., C++ or PROLOG), we focus on highly declarative languages. In fact, many systems and languages for the solution of constraint problems (e.g., AMPL [9], OPL [20], GAMS [5], DLV [7], SMODELS [18], and NP-SPEC [4]) clearly separate the *specification* of a problem, e.g., graph 3-coloring, and its *instance*, e.g., *a* graph, using a two-level architecture for finding solutions: the specification is instantiated (or grounded) against the instance, and then an appropriate solver is invoked. There are several benefits in this separation: obviously declarativeness increases, and the solver is completely decoupled from the specification. Ideally, the programmer can focus only on the specification of the problem, without committing *a priori* to a specific solver. In fact, some systems, e.g., AMPL [9], are able to translate –at the request of the user– a specification in various formats, suitable for different solvers.

Again, the syntax varies a lot among such languages: AMPL, OPL, and GAMS allow the representation of constraints by using algebraic expressions, while DLV, SMODELS, and NP-SPEC are rule-based languages. Anyway, from an abstract point of view, all such languages are extensions of existential second-order logic (ESO) on finite databases, where the existential second-order quantifiers and the first-order formula represent, respectively, the *guess* and *check* phases of the

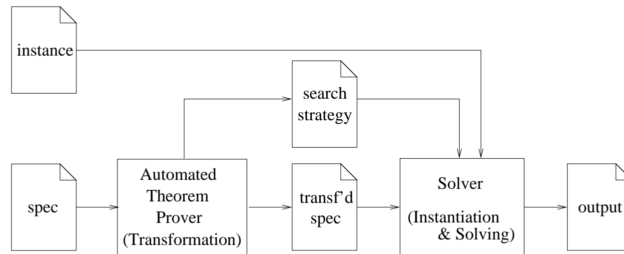


Fig. 1. Architecture of the problem solving system.

constraint modelling paradigm. In particular, in all such languages it is possible to embed ESO queries, and the other way around is also possible, as long as only finite domains are considered.

Since specifications are logical formulae, it is possible to infer their properties by means of automated reasoning tools. The purpose of this paper is exactly to link two important technologies: automated theorem proving (ATP) and constraint programming. The architecture of the system we envision is represented in Figure 1.

In particular, we report the results on using a theorem prover and a finite model finder for reasoning on specifications of constraint problems, represented as ESO formulae. We focus on two forms of reasoning:

– checking existence of *value symmetries*, i.e., properties of the specification that allow to exchange values of the finite domains without losing all solutions; on top of that, we check whether a given formula breaks such symmetries;
– checking existence of *functional dependencies*, i.e., properties that force values of some guessed predicates to depend on the value of some others.

There are at least two reasons why a system should make automatically such checks: first of all, it has been proven that solving can be made much more efficient by, e.g., recognizing and breaking symmetries (a wide literature is nowadays available, cf., e.g., [2, 6]). Secondly, the person performing constraint modelling may be interested in the above properties: as an example, existence (or lack) of a dependency may reveal a bug in the specification.

The main result of this paper is that it is actually possible to use ATP technology to reason on combinatorial problems, and we exhibit several examples proving it. As a side-effect, we propose a new domain of application and a brand new set of benchmarks for ATP systems, which is not represented in large repositories, such as TPTP (cf. `www.tptp.org`).

Relations between constraint satisfaction and deduction have been observed since several years (cf., e.g., the early work [1], and [12] for an up-to-date report). The use of automated tools for preprocessing constraint satisfaction problems (CSPs) has been limited, to the best of our knowledge, to the *instance* level. As an example, the use of packages such as *nauty* [16] for finding symmetries

on CSPs has been proposed in [6]. On the other hand, not much work has been done on reasoning at the *specification* level. A limited form of reasoning is offered by the OPL system, which checks (syntactically) whether a specification contains only linear constraints and objective function, and in this case invokes an integer linear programming solver (typically very efficient); otherwise, it uses a constraint programming solver.

The rest of the paper is organized as follows: in Section 2 we give some preliminaries on modelling combinatorial problems as formulae in ESO. Sections 3 and 4 are devoted to the description of experiments in checking symmetries and dependencies, respectively. In Section 5 we conclude the paper, and present current research.

2 Preliminaries

In this paper, we use *existential second-order logic* (ESO) for the specification of problems, which allows to represent all search problems in the complexity class NP [8]. The use of ESO as a modelling language for problem specifications is common in the database literature, but unusual in constraint programming, therefore few comments are in order. Constraint modelling systems like those mentioned in Section 1 have a richer syntax and more complex constructs, and we plan to eventually move from ESO to such languages. For the moment, we claim that studying the simplified scenario is a mandatory starting point for more complex investigations, and that our results can serve as a basis for reformulating specifications written in higher-level languages. Anyway, examples using the syntax of the implemented language OPL are exhibited in Sections 4 and 5.

Coherently with all state-of-the-art systems, we represent an instance of a problem by means of a *relational database*. All constants appearing in a database are *uninterpreted*, i.e., they don't have a specific meaning.

An ESO specification describing a search problem π is a formula ψ_π

$$\exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R}), \tag{1}$$

where $\boldsymbol{R} = \{R_1, \ldots, R_k\}$ is the relational schema for every input instance (i.e., a fixed set of relations of given arities denoting the schema for all input instances for π), and ϕ is a quantified first-order formula on the relational vocabulary $\boldsymbol{S} \cup \boldsymbol{R} \cup \{=\}$ ("=" is always interpreted as identity).

An instance $\mathcal{I}$ of the problem is given as a relational database over the schema $\boldsymbol{R}$, i.e., as an extension for all relations in $\boldsymbol{R}$. Predicates (of given arities) in the set $\boldsymbol{S} = \{S_1, \ldots, S_n\}$ are called *guessed*, and their possible extensions (with tuples on the domain given by constants occurring in $\mathcal{I}$ plus those occurring in ϕ, i.e., the so called Herbrand universe) encode points in the search space for problem π.

Formula ψ_π correctly encodes problem π if, for every input instance $\mathcal{I}$, a bijective mapping exists between solutions to π and extensions of predicates in

$\boldsymbol{S}$ which verify $\phi(\boldsymbol{S},\mathcal{I})$:

$$\text{For each instance } \mathcal{I}\text{:} \quad \Sigma \text{ is a solution to } \pi(\mathcal{I}) \quad \Longleftrightarrow \quad \{\Sigma,\mathcal{I}\} \models \phi.$$

It is worthwhile to note that, when a specification is instantiated against an input database, a CSP is obtained.

Example 1 (Graph 3-coloring). In this NP-complete decision problem (cf. [10, Prob. GT4, p. 191]) the input is a graph, and the question is whether it is possible to give each of its nodes one out of three colors (red, green, and blue), in such a way that adjacent nodes (not including self-loops) are never colored the same way. The question can be easily specified as an ESO formula ψ on the input schema $\boldsymbol{R} = \{edge(\cdot,\cdot)\}$:

$$\begin{aligned}
\exists RGB \quad & \forall X \;\; R(X) \vee G(X) \vee B(X) \;\wedge && (2)\\
& \forall X \;\; R(X) \rightarrow \neg G(X) \;\wedge && (3)\\
& \forall X \;\; R(X) \rightarrow \neg B(X) \;\wedge && (4)\\
& \forall X \;\; B(X) \rightarrow \neg G(X) \;\wedge && (5)\\
& \forall XY \; X \neq Y \wedge R(X) \wedge R(Y) \rightarrow \neg edge(X,Y) \wedge && (6)\\
& \forall XY \; X \neq Y \wedge G(X) \wedge G(Y) \rightarrow \neg edge(X,Y) \wedge && (7)\\
& \forall XY \; X \neq Y \wedge B(X) \wedge B(Y) \rightarrow \neg edge(X,Y). && (8)
\end{aligned}$$

3 Value symmetries

In this section we face the problem of automatically detecting and breaking some symmetries in problem specifications. In Subsection 3.1 we give preliminary definitions of problem transformation and symmetry taken from [3], and show how the symmetry-detection problem can be reduced to checking semantic properties of first-order formulae. We limit our attention to specifications with monadic guessed predicates only, and to transformations and symmetries on values. Motivations for these limitations are given in [3]; here, we just recall that non-monadic guessed predicates can be transformed in monadic ones by unfolding and by exploiting the finiteness of the input database. We refer to [3] also for considerations on benefits of the technique on the efficiency of problem solving. In Subsection 3.2 we then show how a theorem prover can be used to automatically detect and break symmetries.

3.1 Definitions

Definition 1 (Uniform value transformation (UVT) of a specification [3]). *Given a problem specification $\psi \doteq \exists \boldsymbol{S}\ \phi(\boldsymbol{S},\boldsymbol{R})$, with $\boldsymbol{S} = \{S_1, \ldots S_n\}$, S_i monadic for every $i \in [1,n]$, and input schema $\boldsymbol{R}$, a* uniform value transformation (UVT) *for ψ is a mapping $\sigma : \boldsymbol{S} \rightarrow \boldsymbol{S}$, which is total and onto, i.e., defines a permutation of guessed predicates in $\boldsymbol{S}$.*

The term "uniform value" transformation in Definition 1 is used because swapping monadic guessed predicates is conceptually the same as uniformly exchanging domain values in a CSP.

From here on, given ϕ and σ as in the above definition, ϕ^σ is defined as $\phi[S_1/\sigma(S_1), \ldots, S_n/\sigma(S_n)]$, i.e., ϕ^σ is obtained from ϕ by uniformly substituting every occurrence of each guessed predicate with the one given by the transformation σ. Analogously, ψ^σ is defined as $\exists \boldsymbol{S}\ \phi^\sigma(\boldsymbol{S}, \boldsymbol{R})$.

We now define when a UVT is a symmetry for a given specification.

Definition 2 (Uniform value symmetry (UVS) of a specification [3]). *Let $\psi \doteq \exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R})$, be a specification, with $\boldsymbol{S} = \{S_1, \ldots S_n\}$, S_i monadic for every $i \in [1, n]$, and input schema $\boldsymbol{R}$, and let σ be a UVT for ψ. Transformation σ is a* uniform value symmetry (UVS) *for ψ if every extension for $\boldsymbol{S}$ which satisfies ϕ, satisfies also ϕ^σ and vice versa, regardless of the input instance, i.e., for every extension of the input schema $\boldsymbol{R}$.*

Note that every CSP obtained by instantiating a specification with σ has at least the corresponding uniform value symmetry.

In [3], it is shown that checking whether a UVT is a UVS reduces to checking equivalence of two first-order formulae:

Proposition 1 ([3]). *Let ψ be a problem specification of the kind* (1)*, with only monadic guessed predicates, and σ a UVT for ψ. Transformation σ is a UVS for ψ if and only if $\phi \equiv \phi^\sigma$.*

Once symmetries of a specification have been detected, additional constraints can be added in order to *break* them, i.e., to wipe out from the solution space (some of) the symmetrical points. These kind of constraints are called *symmetry-breaking formulae*, and are defined as follows.

Definition 3 (Symmetry-breaking formula [3]). *Let $\psi \doteq \exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R})$, be a specification, with $\boldsymbol{S} = \{S_1, \ldots S_n\}$, S_i monadic for every $i \in [1, n]$, and input schema $\boldsymbol{R}$, and let σ be a UVS for ψ. A* symmetry-breaking formula *for ψ with respect to symmetry σ is a closed (except for $\boldsymbol{S}$) formula $\beta(\boldsymbol{S})$ such that the following two conditions hold:*

1. *Transformation σ is no longer a symmetry for $\exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R}) \wedge \beta(\boldsymbol{S})$:*
$$(\phi \wedge \beta(\boldsymbol{S})) \not\equiv (\phi \wedge \beta(\boldsymbol{S}))^\sigma\ ;$$
2. *Every model of $\phi(\boldsymbol{S}, \boldsymbol{R})$ can be obtained by those of $\phi(\boldsymbol{S}, \boldsymbol{R}) \wedge \beta(\boldsymbol{S})$ by applying symmetry σ:*
$$\phi(\boldsymbol{S}, \boldsymbol{R})\ \models \bigvee_{\boldsymbol{\sigma} \in \sigma^*} (\phi(\boldsymbol{S}, \boldsymbol{R}) \wedge \beta(\boldsymbol{S}))^{\boldsymbol{\sigma}}. \tag{9}$$
where $\boldsymbol{\sigma}$ is a sequence (of finite length ≥ 0) over σ (i.e., a string in the regular language σ^*), and, given a first-order formula $\gamma(\boldsymbol{S})$, $\gamma(\boldsymbol{S})^{\boldsymbol{\sigma}}$ denotes $(\cdots(\gamma(\boldsymbol{S})^\sigma)\cdots)^\sigma$, i.e., σ is applied $|\boldsymbol{\sigma}|$ times (if $\boldsymbol{\sigma} = \langle\rangle$, then $\gamma(\boldsymbol{S})^{\boldsymbol{\sigma}}$ is $\gamma(\boldsymbol{S})$ itself).

If $\beta(\boldsymbol{S})$ matches the above definition, then we are entitled to solve the problem $\exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R}) \wedge \beta(\boldsymbol{S})$ instead of the original one $\exists \boldsymbol{S}\ \phi(\boldsymbol{S}, \boldsymbol{R})$. In fact, point 1 in the above definition states that formula $\beta(\boldsymbol{S})$ actually breaks σ, since, by Proposition 1, σ is not a symmetry of the rewritten problem. Furthermore, point 2 states that every solution of $\phi(\boldsymbol{S}, \boldsymbol{R})$ can be obtained by repeatedly applying σ to some solutions of $\phi(\boldsymbol{S}, \boldsymbol{R}) \wedge \beta(\boldsymbol{S})$. Hence, all solutions are preserved in the rewritten problem, up to symmetric ones.

It is worthwhile noting that, even if in formula (9) $\boldsymbol{\sigma}$ ranges over the (infinite) set of finite-length sequences of 0 or more applications of σ, this actually reduces to sequences of length at most $n!$, since this is the maximum number of successive applications of σ that can lead to all different permutations. Moreover, we observe that the inverse logical implication always holds, because σ is a UVS, and so $\phi(\boldsymbol{S}, \boldsymbol{R})^{\sigma} \equiv \phi(\boldsymbol{S}, \boldsymbol{R})$.

3.2 Experiments with the theorem prover

Proposition 1 suggests that the problem of detecting UVSs of a specification ψ of the kind (1) can in principle be performed in the following way:

1. Selecting a UVT σ, i.e. a permutation of guessed predicates in ψ (if ψ has n guessed predicates, there are $n!$ such UVTs);
2. Checking whether σ is a UVS, i.e., deciding whether $\phi \equiv \phi^{\sigma}$.

The above procedure suggests that a first-order theorem prover can be used to perform automatically point 2. Even if we proved in [3] that this problem is undecidable, we show how a theorem prover usually performs well on this kind of formulae.

As for the symmetry-breaking problem, from conditions of Definition 3 it follows that also the problem of checking whether a formula breaks a given UVS for a specification clearly reduces to semantic properties of logic formulae.

In this section we give some details about the experimentation done using automated tools. First of all we note that, obviously, all the above conditions can be checked by using a refutational theorem prover. It is interesting to note that, for some of them, we can use a finite model finder. In particular, we can use such a tool for checking statements (such as condition 1 of Definition 3 or the negation of the condition of Proposition 1) which are syntactically a non-equivalence. As a matter of facts, it is enough to look for a finite model of the negation of the statement, i.e., the equivalence. If we find such a model, then we are sure that the non-equivalence holds, and we are done. The tools we used are OTTER [15], and MACE [14], respectively, in full "automatic" mode. Complete source files are available at `http://www.dis.uniroma1.it/~tmancini/research/cilc04`.

Detecting symmetries The examples on which we worked are the following.

Example 2 (Graph 3-coloring: Example 1 continued). The mapping $\sigma^{R,G} : \boldsymbol{S} \rightarrow \boldsymbol{S}$ such that $\sigma^{R,G}(R) = G$, $\sigma^{R,G}(G) = R$, $\sigma^{R,G}(B) = B$ is a UVT for it. It is

easy to observe that formula $\phi^{\sigma^{R,G}}$ is equivalent to ϕ, because clauses of the former are syntactically equivalent to clauses of the latter and vice versa. This implies, by Proposition 1, that $\sigma^{R,G}$ is also a UVS for the specification of the 3-coloring problem. The same happens also for transformations $\sigma^{R,B}$ and $\sigma^{G,B}$ that swap B with, respectively, R and G.

Example 3 (Not-all-equal Sat). In this NP-complete problem [10], the input is a propositional formula in CNF, and the question is whether it is possible to assign a truth value to all the variables in such a way that the input formula is satisfied, and that every clause contains at least one literal whose truth value is false. We assume that the input formula is encoded by the following relations:

- $inclause(\cdot,\cdot)$; tuple $\langle l, c\rangle$ is in *inclause* iff literal l is in clause c;
- $l^+(\cdot,\cdot)$; a tuple $\langle l, v\rangle$ is in l^+ iff l is the positive literal relative to the propositional variable v, i.e., v itself;
- $l^-(\cdot,\cdot)$; a tuple $\langle l, v\rangle$ is in l^- iff l is the negative literal relative to the propositional variable v, i.e., $\neg v$;
- $var(\cdot)$, containing the set of propositional variables occurring in the formula;
- $clause(\cdot)$, containing the set of clauses of the formula.

A specification for this problem is as follows (T and F represent the set of variables whose truth value is true and false, respectively):

$$
\begin{aligned}
\exists TF\ \forall X\ & var(X) \leftrightarrow T(X) \vee F(X) \ \wedge && (10)\\
& \forall X\ \neg\,(T(X) \wedge F(X)) \ \wedge && (11)\\
& \forall C\ clause(C) \rightarrow \\
& \quad \big[\exists L\ inclause(L,C) \wedge \forall V\ \big(l^+(L,V) \rightarrow T(V)\big) \wedge \big(l^-(L,V) \rightarrow F(V)\big)\,\big] \ \wedge && (12)\\
& \forall C\ clause(C) \rightarrow \\
& \quad \big[\exists L\ inclause(L,C) \wedge \forall V\ \big(l^+(L,V) \rightarrow F(V)\big) \wedge \big(l^-(L,V) \rightarrow T(V)\big)\,\big]. && (13)
\end{aligned}
$$

Constraints (10–11) force every variable to be assigned exactly one truth value; moreover, (12) forces the assignment to be a model of the formula, while (13) leaves in every clause at least one literal whose truth value is false.

Let us consider the UVT $\sigma^{T,F}$, defined as $\sigma^{T,F}(T) = F$ and $\sigma^{T,F}(F) = T$. It is easy to prove that $\sigma^{T,F}$ is a UVS for this problem, since $\phi^{\sigma^{T,F}}$ is equivalent to ϕ.

The results we obtained with OTTER are shown in Table 1. The third row refers to the version of the Not-all-equal Sat problem in which all clauses have three literals, the input is encoded using a ternary relation $clause(\cdot,\cdot,\cdot)$, and the specification varies accordingly. It is interesting to see that the performance is always quite good.

A note on the encoding is in order. Initially, we gave the input to OTTER exactly in the format specified by Proposition 1, but the performance was quite poor: for 3-coloring the tool did not even succeed in transforming the formula in

Spec	Symmetry	CPU time (sec)	Proof length	Proof level
3-coloring	$\sigma^{R,G}$	0.27	43	12
Not-all-equal Sat	$\sigma^{T,F}$	0.22	54	19
Not-all-equal 3-Sat	$\sigma^{T,F}$	4.71	676	182

Table 1. Performance of OTTER for proving that a UVT is a UVS.

clausal form, and symmetry was proven only for very simplified versions of the problem, e.g., 2-coloring, omitting constraint (2). Results of Table 1 have been obtained by introducing new propositional variables defining single constraints. As an example, constraint (2) is represented as

```
covRGB <-> (all x (R(x) | G(x) | B(x))).,
```

where `covRGB` is a fresh propositional variable. Obviously, we wrote a first-order logic formula encoding condition of Proposition 1, and gave its negation to OTTER in order to find a refutation.

As for proving non-existence of symmetries, we used the following example.

Example 4 (Graph 3-coloring with red self-loops). We consider a modification of the problem of Example 1, and show that only one of the UVTs in Example 2 is indeed a UVS for the new problem. Here, the question is whether it is possible to 3-color the input graph in such a way that every self loop insists on a red node. In ESO, one more clause (which forces the nodes with self loops to be colored in red) must be added to the specification in Example 1:

$$\forall X \;\; edge(X, X) \rightarrow R(X). \tag{14}$$

UVT $\sigma^{G,B}$ is a UVS also of the new problem, because of the same argument of Example 2. However, for what concerns $\sigma^{R,G}$, in this case $\phi^{\sigma^{R,G}}$ is not equivalent to ϕ: as an example, for the input instance $edge = \{(v, v)\}$, the color assignment $\overline{R}, \overline{G}, \overline{B}$ such that $\overline{R} = \{v\}, \overline{G} = \overline{B} = \emptyset$ is a model for the original problem, i.e., $\overline{R}, \overline{G}, \overline{B} \models \phi(R, G, B, edge)$. It is however easy to observe that $\overline{R}, \overline{G}, \overline{B} \not\models \phi^{\sigma^{R,G}}(R, G, B, \; edge)$, because $\phi^{\sigma^{R,G}}$ is verified only by color assignments for which $\overline{G}(v)$ holds. This implies, by Proposition 1, that $\sigma^{R,G}$ is not a UVS. For the same reason, also $\sigma^{R,B}$ is not a UVS for the new problem.

We wrote a first-order logic formula encoding condition of Proposition 1 for $\sigma^{R,G}$ on the above example and gave its negation to MACE in order to find a model of the non-equivalence. MACE was able to find the model described in Example 4 in less than one second of CPU time.

Breaking symmetries We worked on the 3-coloring problem specification given in Example 1 and the UVS $\sigma^{R,G}$ defined in Example 2. This UVS can be broken in several ways, as an example by the following formula:

$$\beta^{R,G}_{sel}(R,G,B) \doteq R(\overline{v}) \vee B(\overline{v}), \tag{15}$$

that forces a selected node, say $\overline{v}$, not to be colored in green. The simpler formula $R(\overline{v})$ breaks two symmetries, namely $\sigma^{R,G}$ and $\sigma^{R,B}$, and can be obtained as the logical and of (15) and $R(\overline{v}) \vee G(\overline{v})$.

We used MACE and OTTER in order to prove that (15) is indeed a symmetry-breaking formula for the 3-coloring problem specification with respect to $\sigma^{R,G}$, i.e., for testing conditions 1 and 2 (respectively) of Definition 3. Both systems succeeded in less than one second of CPU time.

As described in [3], a UVS can be broken in several ways, and with different effectiveness. As an example, $\sigma^{R,G}$ in the 3-coloring problem specification can be broken also by the following formula:

$$\beta^{R,G}_{card}(R,G,B) \ \doteq \ |R| \leq |G|, \tag{16}$$

that forces green nodes to be at least as many as red ones. It is easy to prove that formula (16) respects both conditions of Definition 3, and of course it breaks the symmetry more effectively than formula (15), since formulae at the two sides of condition 1 of Definition 3 have few common models (cf. [3] for a discussion of the *effectiveness* of a symmetry-breaking formula. It is worth noting that this concept alludes to how completely a formula breaks the symmetry, and it is not related to efficiency issues, e.g., the amenability of the constraint to be propagated.)

However, this example highlights some difficulties that can arise when using first-order ATPs. In fact, although constraint (16) can be written in ESO using standard techniques, it is not first-order definable. Therefore, conditions in Definition 3 are (non-)equivalence of second-order formulae. So, the use of a first-order theorem prover may in general not suffice.

However, in some circumstances, it is possible to synthesize first-order conditions that can be used to infer the truth value of those of Definition 3. This is the case of formulae defined in ESO. As an example, by using MACE and OTTER collaboratively, we proved point 1 of Definition 3 for formula (16) on the 3-coloring problem specification in few hundredths of second.

4 Dependent predicates

In this section we tackle the problem of recognizing guessed predicates that functionally depend on the others in a given specification. This means that, for every solution of any instance, the extension of a dependent guessed predicate is determined by the extensions of the others.

Recognizing functionally dependent predicates in a specification is very important for the efficiency of any backtracking solver, since branches regarding dependent predicates (that represent values assigned to variables of the CSP obtained after instantiation) can be safely avoided. As an example, it is shown in [11] how to modify the Davis-Putnam procedure for SAT so that is avoids

branches on variables added during the clausification of non-CNF formulae, since values assigned to these variables depend on assignments to the other ones. Moreover, specific SAT solvers, e.g., EQSATZ [13], have been developed in order to appropriately handle (by means of the so-called "equivalence reasoning") equivalence clauses, which have been recognized to be a very common structure in the SAT encoding of many hard real-world problems, and a major obstacle to the Davis-Putnam procedure.

The automatic recognition of functionally dependent predicates is important also to improve the quality aspects of specifications as software artefacts. A dependent predicate may be an evidence either of a bug in the problem model, or of a bad design choice, since the adopted model for the problem is, in some sense, redundant. Of course, the use of dependent predicates can also be the consequence of a precise design choice, e.g., with the goal of a more modular and readable problem specification. In any case, a feature of the system that automatically checks whether a dependence holds is likely to be useful to the designer.

In Subsection 4.1, we give the formal definition of dependent predicates in a specification, and in Subsection 4.2 show how the problem of checking whether a set of guessed predicates is dependent from the others reduces to check semantic properties of a first-order formula. We observe that definitions and results in this section are original, and do not appear elsewhere. Proof of theorems are not given for space reasons, and will appear in the full paper.

4.1 Definitions

Definition 4 (Functional dependence of a set of predicates in a specification). *Given a problem specification $\psi \doteq \exists \boldsymbol{S}\boldsymbol{P}\ \phi(\boldsymbol{S}, \boldsymbol{P}, \boldsymbol{R})$, with input schema $\boldsymbol{R}$, $\boldsymbol{P}$* functionally depends *on $\boldsymbol{S}$ if, for each instance $\mathcal{I}$ of $\boldsymbol{R}$ and for each pair of interpretations M, N of $(\boldsymbol{S}, \boldsymbol{P})$ it holds that, if*

1. *$M \neq N$, and*
2. *$M, \mathcal{I} \models \phi$, and*
3. *$N, \mathcal{I} \models \phi$,*

then $M_{|\boldsymbol{S}} \neq N_{|\boldsymbol{S}}$, where $\cdot_{|\boldsymbol{S}}$ denotes the restriction of an interpretation to predicates in $\boldsymbol{S}$.

The above definition states that $\boldsymbol{P}$ functionally depends on $\boldsymbol{S}$, or that $\boldsymbol{S}$ functionally determines $\boldsymbol{P}$, if it is the case that, regardless of the instance, each pair of distinct solutions of ψ must differ on predicates in $\boldsymbol{S}$, which is equivalent to say that no two different solutions of ψ exist that coincide on the extension for predicates in $\boldsymbol{S}$ but differ on that for predicates in $\boldsymbol{P}$.

Example 5 (Graph 3-coloring: Example 1 continued). In the 3-coloring problem, one of the three guessed predicates is functionally dependent on the others. As an example, B functionally depends on R and G, since, regardless of the instance, it can be defined as $\forall X\ B(X) \leftrightarrow \neg(R(X) \vee G(X))$: constraint (2) is

equivalent to $\forall X\ \neg(R(X) \vee G(X)) \rightarrow B(X)$ and (4) and (5) imply $\forall X\ B(X) \rightarrow \neg(R(X) \vee G(X))$. In other words, for every input instance, no two different solutions exist that coincide on the set of red and green nodes, but differ on the set of blue ones.

Example 6 (Not-all-equal Sat: Example 3 continued). One of the two guessed predicates T and F is functionally dependent on the other, since by constraints (10–11) it follows, e.g., $\forall X\ F(X) \leftrightarrow var(X) \wedge \neg T(X)$.

Next, we show that the problem of checking whether a subset of the guessed predicates in a specification is functionally dependent on the remaining ones, reduces to verifying semantic properties of a first-order formula. To simplify notations, given a list of predicates $\boldsymbol{T}$, we write $\boldsymbol{T'}$ for representing a list of the same number of predicates with, respectively, the same arities, that are fresh, i.e., do not occur elsewhere in the context at hand. Also, $\boldsymbol{T} \equiv \boldsymbol{T'}$ will be a shorthand for the formula

$$\bigwedge_{T \in \boldsymbol{T}} \forall \boldsymbol{X}\ T(\boldsymbol{X}) \equiv T'(\boldsymbol{X}),$$

where T and T' are corresponding predicates in $\boldsymbol{T}$ and $\boldsymbol{T'}$, respectively, and $\boldsymbol{X}$ is a list of variables of the appropriate arity.

Theorem 1. *Let $\psi \doteq \exists \boldsymbol{S}\boldsymbol{P}\ \phi(\boldsymbol{S}, \boldsymbol{P}, \boldsymbol{R})$ be a problem specification with input schema $\boldsymbol{R}$. $\boldsymbol{P}$ functionally depends on $\boldsymbol{S}$ if and only if the following formula is valid:*

$$[\phi(\boldsymbol{S}, \boldsymbol{P}, \boldsymbol{R}) \wedge \phi(\boldsymbol{S'}, \boldsymbol{P'}, \boldsymbol{R}) \wedge \neg(\boldsymbol{S}\boldsymbol{P} \equiv \boldsymbol{S'}\boldsymbol{P'})] \rightarrow \neg(\boldsymbol{S} \equiv \boldsymbol{S'}). \tag{17}$$

Unfortunately, the problem of checking whether the set of predicates in $\boldsymbol{P}$ is functionally dependent on the set $\boldsymbol{S}$ is undecidable, as the following result shows:

Theorem 2. *Given a specification on input schema $\boldsymbol{R}$, and a partition $(\boldsymbol{S}, \boldsymbol{P})$ of its guessed predicates, the problem of checking whether $\boldsymbol{P}$ functionally depends on $\boldsymbol{S}$ is not decidable.*

Nonetheless, as shown in the next section, an ATP usually performs very well in deciding whether formulae of the kind of (17) are valid or not.

4.2 Experiments with the theorem prover

Using Theorem 1 it is easy to write a first-order formula that is valid if and only if a given dependency holds. We used OTTER for proving the existence of dependencies among guessed predicates of different problem specifications:

- Graph 3-coloring (cf. Example 5), where one among the guessed predicates R, G, B is dependent on the others.
- Not-all-equal Sat (cf. Example 6), where one between the guessed predicates T and F is dependent on the other.

For each of the above specifications, we wrote a first-order logic encoding of formula (17), and gave its negation to OTTER in order to find a refutation.

For the purpose of testing effectiveness of the proposed technique in the context of specifications written in implemented languages, we considered also the *Sailco inventory* problem, taken from the OPL book [20, Section 9.4, Statement 9.17] and part of the OPLSTUDIO distribution package (as file `sailco.mod`).

Example 7 (The Sailco inventory problem). This problem specification models a simple inventory application, in which the question is to decide how many sailboats the Sailco company has to produce over a given number of time periods, in order to satisfy the demand and to minimize production costs. The demand for the periods is known and, in addition, an `inventory` of boats is available initially. In each period, Sailco can produce a maximum number of boats (`capacity`) at a given unitary cost (`regularCost`). Additional boats can be produced, but at higher cost (`extraCost`). Storing boats in the inventory also has a cost per period (`inventoryCost` per boat).

Figure 2 shows an OPL model for this problem. An equivalent –apart, of course, for the objective function– ESO specification would be more complex, because of the presence of arithmetic operations in the constraints, and thus will not be presented. However, the analogous of the instance relational schema, guessed predicates, and constraints can be clearly distinguished in the OPL code. Guessed predicates of the ESO specification can be obtained in standard ways: as an example, for the inventory we can define a guessed predicate $inv(\cdot,\cdot)$ with the first argument being the period, and the second one the amount of boats stored in that period, plus additional constraints to force exactly one tuple to belong to $inv(\cdot,\cdot)$ for each period.

From the specification in Figure 2, it can be observed that the amount of boats in the inventory for each time period `t` $>$ 0 (i.e., `inv[t]`) is defined in terms of the amount of regular and extra boats produced in period `t` by the following relationship: `inv[t] = regulBoat[t] + extraBoat[t] - demand[t] + inv[t-1]`. Of course, the same relationship holds in the equivalent ESO specification, making predicate $inv(\cdot,\cdot)$ functionally dependent on $regulBoat(\cdot,\cdot)$ and $extraBoat(\cdot,\cdot)$.

We opted for an OTTER encoding that uses function symbols: as an example, the `inv[]` array is translated to a function symbol $inv(\cdot)$ rather than to a binary predicate (the second argument being the time point). More precisely, according to Theorem 1, a pair of function symbols $inv(\cdot)$ and $inv'(\cdot)$ is introduced. The same happens for `regulBoat[]` and `extraBoat[]`. Moreover, we included in the OTTER formula the following formulae which allow to infer $\forall t\ inv(t) = inv'(t)$ from equality of inv and inv' at the initial time period and equivalence of increments in all time intervals of length 1.

```
equalDiscrete <-> (inv(0) = inv1(0) &
                   (all t (t > 0 -> (inv(t) - inv(t-1)) =
                                     (inv1(t) - inv1(t-1))))).
induction <-> (equalDiscrete -> (all t (inv(t) = inv1(t)))).
```

```
// Instance schema
int+ nbPeriods = ...;            range Periods 1..nbPeriods;

float+ demand[Periods] = ...;    float+ regularCost = ...;
float+ extraCost = ...;          float+ capacity = ...;
float+ inventory = ...;          float+ inventoryCost = ...;

// Guessed predicates
var float+ regulBoat[Periods];   var float+ extraBoat[Periods];
var float+ inv[0..nbPeriods];

// Objective function
minimize ...

// Constraints
subject to {
   inv[0] = inventory;
   forall(t in Periods) regulBoat[t] <= capacity;
   forall(t in Periods)
      regulBoat[t] + extraBoat[t] + inv[t-1] = inv[t] + demand[t];
};
```

Fig. 2. OPL specification for the Sailco problem.

Spec	$\boldsymbol{S}$	$\boldsymbol{P}$	CPU time (sec)	Proof length	Proof level
3-coloring	R, G	B	0.25	27	18
Not-all-equal 3-Sat	T	F	0.38	18	14
Sailco	$regulBoat$, $extraBoat$	inv	0.21	29	11

Table 2. Performance of OTTER for proving that the set $\boldsymbol{P}$ of guessed predicates is functionally dependent on the set $\boldsymbol{S}$.

Results of the experiments are presented in Table 2. As it can be observed, the time needed by OTTER is always very low.

5 Conclusions and current research

The use of automated tools for preprocessing CSPs has been limited, to the best of our knowledge, to the instance level (cf. Section 1 for references). In this paper we proved that current ATP technology is able to perform significant forms of reasoning on specifications of constraint problems. We focused on two forms of reasoning: symmetry detection and breaking, and functional dependence checking. Reasoning has been done for various problems, including the ESO encodings of graph 3-coloring and Not-all-equal Sat, and the OPL encoding of an inventory problem. In general, reasoning is done very efficiently by the ATP, although

```
int+ N = ...;
range Row 1..N;  range Col 1..N;
var Col Queen[Row];
solve {
   forall (r1, r2 in Row : r1 <> r2) {
      Queen[r1] <> Queen[r2];           // no vertical attack
      Queen[r1] + r1 <> Queen[r2] + r2;  // no NW-SE diagonal attack
      Queen[r1] - r1 <> Queen[r2] - r2;  // no NE-SW diagonal attack
}};
```

Fig. 3. OPL specification for the N-queens problem.

effectiveness depends on the format of the input, and auxiliary propositional variables seem to be necessary.

There are indeed some tasks, namely, proving existence of symmetries in the *Social golfer problem* (problem 10 at `www.csplib.org`) which OTTER –in the automatic mode– was unable to do. So far, we used only two tools, namely OTTER and MACE, and plan to investigate effectiveness of other provers, e.g., VAMPIRE [19]. We note that the wide availability of constraint problem specifications, both in implemented languages, cf., e.g., [9, 20], and in natural language, cf., e.g., [10], the CSP-Library (`www.csplib.org`), the OR-Library (`www.ms.ic.ac.uk/info.html`), offers a brand new set of benchmarks for ATP systems, which is not represented in large repositories, such as TPTP (cf. `www.tptp.org`).

We believe that ATPs can be used also for other useful forms of reasoning, apart from those described in this paper. As an example, in Figure 3 we show the OPL specification of the N-queens problem, cf. [20, Section 2.2, Statement 2.16], which states that three constraints must hold for all pairs of distinct rows (cf. the condition `r1 <> r2`). For symmetry reasons, a solution-preserving (and possibly more efficient) formulation requires the constraints to hold just for *totally ordered* pairs of rows, i.e., `r1 < r2`. From the logical point of view, this can be recognized simply by proving that swapping `r1` and `r2` leads to an equivalent specification. OTTER was able to prove such an equivalence in less than one second of CPU time. We are currently investigating the applicability of such a technique for a general class of specifications, in which symmetries on *variables* [17] hold.

Acknowledgements This research has been supported by MIUR (Italian Ministry for Instruction, University, and Research) under the FIRB project ASTRO (Automazione dell'Ingegneria del Software basata su Conoscenza), and under the COFIN project "Design and development of a software system for the specification and efficient solution of combinatorial problems, based on a high-level language, and techniques for intensional reasoning and local search". The authors are indebted to Marco Schaerf for fruitful discussions, and in particular for Definition 4.

References

1. W. Bibel. Constraint satisfaction from a deductive viewpoint. *Artificial Intelligence*, 35:401–413, 1988.
2. C. A. Brown, L. Finkelstein, and P. W. Purdom. Backtrack searching in the presence of symmetry. In T. Mora, editor, *Proc. of 6th Intl. Conf. on Applied Algebra, Algebraic Algorithms and Error Correcting codes*, pages 99–110. Springer, 1988.
3. M. Cadoli and T. Mancini. Detecting and breaking symmetries on specifications. In *Proc. of SymCon'03 (CP 2003)*, 2003. Available at `http://scom.hud.ac.uk/scombms/SymCon03/Papers/Cadoli.pdf`.
4. M. Cadoli and A. Schaerf. Compiling problem specifications into SAT. In *Proc. of ESOP'01*, vol. 2028 of *LNCS*, pages 387–401. Springer, 2001.
5. E. Castillo, A. J. Conejo, P. Pedregal, and N. A. Ricardo Garca. *Building and Solving Mathematical Programming Models in Engineering and Science*. Wiley, 2001.
6. J. M. Crawford, M. L. Ginsberg, E. M. Luks, and A. Roy. Symmetry-breaking predicates for search problems. In *Proc. of KR'96*, pages 148–159, 1996.
7. T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The KR system `dlv`: Progress report, Comparisons and Benchmarks. In *Proc. of KR'98*, pages 406–417, 1998.
8. R. Fagin. Generalized First-Order Spectra and Polynomial-Time Recognizable Sets. In R. M. Karp, editor, *Complexity of Computation*, pages 43–74. AMS, 1974.
9. R. Fourer, D. M. Gay, and B. W. Kernigham. *AMPL: A Modeling Language for Mathematical Programming*. International Thomson Publishing, 1993.
10. M. R. Garey and D. S. Johnson. *Computers and Intractability—A guide to NP-completeness*. W.H. Freeman and Company, San Francisco, 1979.
11. E. Giunchiglia and R. Sebastiani. Applying the Davis-Putnam procedure to non-clausal formulas. In *Proc. of AI*IA'99*, volume 1792 of *LNAI*, pages 84–94. Springer, 2000.
12. P. G. Kolaitis. Constraint satisfaction, databases, and logic. In *Proc. of IJCAI'03*, pages 1587–1595, 2003.
13. C. M. Li. Integrating equivalency reasoning into Davis-Putnam procedure. In *Proc. of AAAI'00*. AAAI / MIT Press, 2000.
14. W. McCune. Mace 2.0 reference manual and guide. Technical Report ANL/MCS-TM-249, Argonne National Laboratory, Mathematics and Computer Science Division, May 2001. Available at `http://www-unix.mcs.anl.gov/AR/mace/`.
15. W. McCune. Otter 3.3 reference manual. Technical Report ANL/MCS-TM-263, Argonne National Laboratory, Mathematics and Computer Science Division, August 2003. Available at `http://www-unix.mcs.anl.gov/AR/otter/`.
16. B. D. McKay. *nauty* user's guide (version 2.2). Available at `http://cs.anu.edu.au/~bdm/nauty/nug.pdf`, 2003.
17. P. Meseguer and C. Torras. Solving strategies for highly symmetric CSPs. In *Proc. of IJCAI'99*, pages 400–405, 1999.
18. I. Niemelä. Logic programs with stable model semantics as a constraint programming paradigm. *Ann. of Mathematics and Artificial Intelligence*, 25(3,4):241–273, 1999.
19. A. Riazanov and A. Voronkov. Vampire. In *Proc. of CADE'99*, volume 1632 of *LNAI*, 1999.
20. P. Van Hentenryck. *The OPL Optimization Programming Language*. The MIT Press, 1999.

Constrained CP-nets

Steve Prestwich[1], Francesca Rossi[2], Kristen Brent Venable[2], Toby Walsh[1]

1: Cork Constraint Computation Centre, University College Cork, Ireland. Email: s.prestwich@cs.ucc.ie, tw@4c.ucc.ie
2: Department of Pure and Applied Mathematics, University of Padova, Italy. Email: {frossi,kvenable}@math.unipd.it.

Abstract. We present a novel approach to deal with preferences expressed as a mixture of hard constraints, soft constraints, and CP-nets. We construct a set of hard constraints whose solutions are the optimal solutions of the set of preferences, where optimal is defined differently w.r.t. other approaches [2, 7]. The new definition of optimality introduced in this paper, allows us to avoid dominance testing (is one outcome better than another?) which is a very expensive operation often used when finding optimal solutions or testing optimality, while being reasonable and intuitive. We also show how hard constraints can sometimes eliminate cycles in the preference ordering. Finally, we extend this approach to deal with the preferences of multiple agents. This simple and elegant technique permits conventional constraint and SAT solvers to solve problems involving both preferences and constraints.

1 Introduction

Preferences and constraints are ubiquitous in real-life scenarios. We often have hard constraints (as “I must be at the office before 9am”) as well as some preferences (as “I would prefer to be at the office around 8:30am” or “I would prefer to go to work by bicycle rather than by car”). Whilst formalisms to represent and reason about hard constraints are relatively stable, having been studied for over 20 years [5], preferences have not received as much attention until more recent times. Among the many existing approaches to represent preferences, we will consider CP-nets [6, 3], which is a qualitative approach where preferences are given by ordering outcomes (as in “I like meat over fish”) and soft constraints [1], which is a quantitative approach where preferences are given to each statement in absolute terms (as in “My preference for fish is 0.5 and for meat is 0.9”).

It is easy to reason with hard and soft constraints at the same time, since hard constraints are just a special case of soft constraints. Much less is understood about reasoning with CP-nets and (hard or soft) constraints. One of our aims is to tackle this problem. We will define a structure called a constrained CP-net. This is just a CP-net plus a set of hard constraints. We will give a semantics for this structure (based on the original flipping semantics of CP-nets) which gives priority to the hard constraints. We will show how to obtain the optimal solutions of such a constrained CP-net by compiling the preferences into a set of hard constraints whose solutions are exactly the optimal solutions of the constrained CP-net. This allows us to test optimality in linear

time, even if the CP-net is not acyclic. Finding an optimal solution of a constrained CP net is NP-hard [1] (as it is in CP-nets and in hard constraints).

Prior to this work, to test optimality of a CP-net plus a set of constraints, we had to find all solutions of the constraints (which is NP-hard) and then test if any of them dominate the solution in question [2]. Unfortunately dominance testing is not known to be in NP even for acyclic CP-nets, as we may have to explore chains of worsening flips that are exponentially long. By comparison, we do not need to perform dominance testing in our approach. Our semantics is also useful when the CP-net defines a preference ordering that contains cycles, since the hard constraints can eliminate these cycles. Lastly, since we compile preferences down into hard constraints, we can use standard constraint solving algorithms (or SAT algorithms if the variables have just two values) to reason about preferences and constraints, rather than develop special purpose algorithms for constrained CP-nets (as in [2]). We also consider when a CP-net is paired with a set of soft constraints, and when there are several CP-nets, and sets of hard or soft constraints. In all these cases, optimal solutions can be found by solving a set of hard or soft constraints, avoiding dominance testing.

2 Background

2.1 CP-nets

In many applications, it is natural to express preferences via generic qualitative (usually partial) preference relations over variable assignments. For example, it is often more intuitive to say "I prefer red wine to white wine", rather than "Red wine has preference 0.7 and white wine has preference 0.4". The former statement provides less information, but does not require careful selection of preference values. Moreover, we often wish to represent conditional preferences, as in "If it is meat, then I prefer red wine to white". Qualitative and conditional preference statements are thus useful components of many applications.

CP-nets [6, 3] are a graphical model for compactly representing conditional and qualitative preference relations. They exploit conditional preferential independence by structuring an agent's preferences under the *ceteris paribus* assumption. Informally, CP-nets are sets of *conditional ceteris paribus (CP)* preference statements. For instance, the statement *"I prefer red wine to white wine if meat is served."* asserts that, given two meals that differ *only* in the kind of wine served *and* both containing meat, the meal with a red wine is preferable to the meal with a white wine. Many users' preferences appear to be of this type.

CP-nets bear some similarity to Bayesian networks. Both utilize directed graphs where each node stands for a domain variable, and assume a set of features $F = \{X_1, \ldots, X_n\}$ with finite domains $\mathcal{D}(X_1), \ldots, \mathcal{D}(X_n)$. For each feature X_i, each user specifies a set of *parent* features $Pa(X_i)$ that can affect her preferences over the values of X_i. This defines a dependency graph in which each node X_i has $Pa(X_i)$ as

[1] More precisely it is in FNP-hard, since it is not a decision problem. In the rest of the paper we will write NP meaning FNP when not related to decision problems.

its immediate predecessors. Given this structural information, the user explicitly specifies her preference over the values of X_i for *each complete outcome* on $Pa(X_i)$. This preference is assumed to take the form of total or partial order over $\mathcal{D}(X)$ [6, 3].

For example, consider a CP-net whose features are A, B, C, and D, with binary domains containing f and $\overline{f}$ if F is the name of the feature, and with the preference statements as follows: $a \succ \overline{a}$, $b \succ \overline{b}$, $(a \wedge b) \vee (\overline{a} \wedge \overline{b}) : c \succ \overline{c}$, $(a \wedge \overline{b}) \vee (\overline{a} \wedge b) : \overline{c} \succ c$, $c : d \succ \overline{d}$, $\overline{c} : \overline{d} \succ d$. Here, statement $a \succ \overline{a}$ represents the unconditional preference for $A = a$ over $A = \overline{a}$, while statement $c : d \succ \overline{d}$ states that $D = d$ is preferred to $D = \overline{d}$, given that $C = c$.

The semantics of CP-nets depends on the notion of a worsening flip. A worsening flip is a change in the value of a variable to a value which is less preferred by the CP statement for that variable. For example, in the CP-net above, passing from $abcd$ to $ab\overline{c}d$ is a worsening flip since c is better than $\overline{c}$ given a and b. We say that one outcome α is better than another outcome β (written $\alpha \succ \beta$) iff there is a chain of worsening flips from α to β. This definition induces a strict partial order over the outcomes. In general, there may be many optimal outcomes. However, in acyclic CP-nets (that is, CP-nets with an acyclic dependency graph), there is only one.

Several types of queries can be asked about CP-nets. First, given a CP-net, what are the optimal outcomes? For acyclic CP-nets, such a query is answerable in linear time [6, 3]: we forward sweep through the CP-net, starting with the unconditional variables, following the arrows in the dependency graph and assigning at each step the most preferred value in the preference table. For instance, in the CP-net above, we would choose $A = a$ and $B = b$, then $C = c$ and then $D = d$. The optimal outcome is therefore $abcd$. The same complexity also holds for testing whether an outcome is optimal since an acyclic CP-net has only one optimal outcome. We can find this optimal outcome (in linear time) and then compare it to the given one (again in linear time). On the other hand, for cyclic CP-nets, both finding and testing optimal outcomes is NP-hard.

The second type of query is a dominance query. Given two outcomes, is one better than the other? Unfortunately, this query is NP-hard even for acyclic CP-nets. Whilst tractable special cases exist, there are also acyclic CP-nets in which there are exponentially long chains of worsening flips between two outcomes. In the CP-net of the example, $\overline{a}b\overline{c}\overline{d}$ is worse than $abcd$.

2.2 Soft and hard constraints

There are several formalisms for describing *soft constraints*. We use the c-semi-ring formalism [1] as this generalizes most of the others. In brief, a soft constraint associates each instantiation of its variables with a value from a partially ordered set. We also supply operations for combining ($\times$) and comparing (+) values. A semi-ring is a tuple $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ such that: A is a set and $\mathbf{0}, \mathbf{1} \in A$; $+$ is commutative, associative and $\mathbf{0}$ is its unit element; $\times$ is associative, distributes over $+$, $\mathbf{1}$ is its unit element and $\mathbf{0}$ is its absorbing element. A *c-semi-ring* is a semi-ring $\langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$ in which $+$ is idempotent, $\mathbf{1}$ is its absorbing element and $\times$ is commutative.

Let us consider the relation $\leq$ over A such that $a \leq b$ iff $a + b = b$. Then $\leq$ is a partial order, $+$ and $\times$ are monotone on $\leq$, $\mathbf{0}$ is its minimum and $\mathbf{1}$ its maximum, $\langle A, \leq \rangle$ is a complete lattice and, for all $a, b \in A$, $a + b = lub(a, b)$. Moreover, if $\times$ is

idempotent: $+$ distributes over $\times$; $\langle A, \leq \rangle$ is a complete distributive lattice and $\times$ its glb. Informally, the relation $\leq$ compares semi-ring values and constraints. When $a \leq b$, we say that *b is better than a*. Given a semi-ring $S = \langle A, +, \times, \mathbf{0}, \mathbf{1} \rangle$, a finite set D (variable domains) and an ordered set of variables V, a *soft constraint* is a pair $\langle def, con \rangle$ where $con \subseteq V$ and $def : D^{|con|} \rightarrow A$. A constraint specifies a set of variables, and assigns to each tuple of values of these variables an element of the semi-ring.

A *soft constraint satisfaction problem* (SCSP) is given by a set of soft constraints. A solution to an SCSP is a complete assignment to its variables, and the preference value associated with a solution is obtained by multiplying the preference values of the projections of the solution to each constraint. A solution is better than another if its preference value is higher in the partial order of the semi-ring. Finding an optimal solution for an SCSP is NP-hard. On the other hand, given two solutions, checking whether one is preferable to another is straightforward: compute the semi-ring values of the two solutions and compare the resulting two values.

Each semiring identifies a class of soft constraints. For example, fuzzy CSPs are SCSPs over the semiring $S_{FCSP} = \langle [0, 1], max, min, 0, 1 \rangle$. This means that preferences are over [0,1], and that we want to maximize the minimum preference over all the constraints. Another example is given by weighted CSPs, which are just SCSPs over the semiring $S_{weight} = \langle \mathcal{R}, min, +, 0, +\infty \rangle$, which means that preferences (better called costs here) are real numbers, and that we want to minimize their sum.

Note that hard constraints are just a special class of soft constraints: those over the semiring $S_{CSP} = \langle \{false, true\}, \vee, \wedge, false, true \rangle$, which means that there are just two preferences ($false$ and $true$), that the preference of a solution is the logical *and* of the preferences of their subtuples in the constraints, and that true is better than false (ordering induced by the logical *or* operation $\vee$).

3 Constrained CP-nets

We now define a structure which is a CP-net plus a set of hard constraints. In later sections we will relax this concept by allowing soft constraints rather than hard constraints.

Definition 1 (constrained CP-net). *A Constrained CP-net is a CP-net plus some constraints on subsets of its variables. We will thus write a constrained CP-net as a pair $\langle N, C \rangle$, where N is a set of conditional preference statements defining a CP-net and C is a set of constraints.*

The hard constraints can be expressed by generic relations on partial assignments or, in the case of binary features, by a set of Boolean clauses. As with CP-nets, the basis of the semantics of constrained CP-nets is the preference ordering, $\succ$, which is defined by means of the notion of a worsening flip. A worsening flip is defined very similarly to how it is defined in a regular (unconstrained) CP-net.

Definition 2 ($O1 \succ O_2$). *Given a constrained CP-net $\langle N, C \rangle$, outcome O_1 is* **better** *than outcome O_2 (written $O_1 \succ O_2$) iff there is a chain of flips from O_1 to O_2, where each flip is worsening for N and each outcome in the chain satisfies C.*

The only difference with the semantics of (unconstrained) CP-nets is that we now restrict ourselves to chains of *feasible* outcomes. As we show shortly, this simple change has some very beneficial effects. First, we observe that the $\succ$ relation remains a strict partial ordering as it was for CP-nets [6, 3]. Second, it is easy to see that checking if an outcome is optimal is linear (we merely need to check it is feasible and any flip is worsening). Third, if a set of hard constraints are satisfiable and a CP-net is acyclic, then the constrained CP-net formed from putting the hard constraints and the CP-net together must have at least one feasible and undominated outcome. In other words, adding constraints to an acyclic CP-net does not eliminate all the optimal outcomes (unless it eliminates all outcomes). Compare this to [2] where adding constraints to a CP-net may make all the undominated outcomes infeasible while not allowing any new outcomes to be optimal. For example, if we have $O_1 \succ O_2 \succ O_3$ in a CP-net, and the hard constraints make O_1 infeasible, then according to our semantics O_2 is optimal, while according to the semantics in [2] no feasible outcome is optimal.

Theorem 1. *A constrained and acyclic CP-net either has no feasible outcomes or has at least one feasible and undominated outcome.*

Proof. Take an acyclic constrained CP-net $\langle N, C\rangle$. N induces a preference ordering that contains no cycles and has exactly one most preferred outcome, say O. If O is feasible, it is optimal for $\langle N, C\rangle$. If O is infeasible, we move down the preference ordering until at some point we hit the first feasible outcome. This is optimal for $\langle N, C\rangle$. □

4 An example

We will illustrate constrained CP-nets by means of a simple example. This example illustrates that adding constraints can eliminate cycles in the preference ordering defined by the CP-net. This is not true for the semantics of [6], where adding hard constraints cannot break cycles.

Suppose I want to fly to Australia. I can fly with British Airways (BA) or Singapore Airlines, and I can choose between business or economy. If I fly Singapore, then I prefer to save money and fly economy rather than business as there is good leg room even in economy. However, if I fly BA, I prefer business to economy as there is insufficient leg room in their economy cabin. If I fly business, then I prefer Singapore to BA as Singapore's inflight service is much better. Finally, if I have to fly economy, then I prefer BA to Singapore as I collect BA's airmiles. If we use a for British Airways, $\overline{a}$ for Singapore Airlines, b for business, and $\overline{b}$ for economy then we have: $a : b \succ \overline{b}$, $\overline{a} : \overline{b} \succ b$, $b : \overline{a} \succ a$, and $\overline{b} : a \succ \overline{a}$.

This CP-net has chains of worsening flips which contain cycles. For instance, $ab \succ a\overline{b} \succ \overline{a}\overline{b} \succ \overline{a}b \succ ab$. That is, I prefer to fly BA in business (ab) than BA in economy ($a\overline{b}$) for the leg room, which I prefer to Singapore in economy ($\overline{a}\overline{b}$) for the airmiles, which I prefer to Singapore in business ($\overline{a}b$) to save money, which I prefer to BA in business (ab) for the inflight service. According to the semantics of CP-nets, none of the outcomes in the cycle is optimal, since there is always another outcome which is better.

Suppose now that my travel budget is limited, and that whilst Singapore offers no discounts on their business fares, I have enough airmiles with BA to upgrade from economy. I therefore add the constraint that, whilst BA in business is feasible, Singapore in business is not. That is, $\overline{a}b$ is not feasible. In this constrained CP-net, according to our new semantics, there is no cycle of worsening flips as the hard constraints break the chain by making $\overline{a}b$ infeasible. There is one feasible outcome that is undominated, that is, ab. I fly BA in business using my airmiles to get the upgrade. I am certainly happy with this outcome.

Notice that the notion of optimality introduced in this paper gives priority to the constraints with respect to the CP-net. In fact, an outcome is optimal if it is feasible and it is undominated in the constrained CP-net ordering. Therefore, while it is not possible for an infeasible outcome to be optimal, it is possible for an outcome which is dominated in the CP-net ordering to be optimal in the constrained CP-net.

5 Finding optimal outcomes

We now show how to map any constrained CP-net onto an equivalent constraint satisfaction problem containing just hard constraints, such that the solutions of these hard constraints corresponds to the optimal outcomes of the constrained CP-net. The basic idea is that each conditional preference statement of the given CP-net maps onto a conditional hard constraint. For simplicity, we will first describe the construction for Boolean variables. In the next section, we will pass to the more general case of variables with more than two elements in their domain.

Consider a constrained CP-net $\langle N, C\rangle$. Since we are dealing with Boolean variables, the constraints in C can be seen as a set of Boolean clauses, which we will assume are in conjunctive normal form. We now define the **optimality constraints** for $\langle N, C\rangle$, written as $N \oplus_b C$ where the subscript b stands for Boolean variables, as $C \cup \{opt_C(p) \mid p \in N\}$. The function opt maps the conditional preference statement $\varphi : a \succ \overline{a}$ onto the hard constraint:

$$(\varphi \wedge \bigwedge_{\psi \in C, \overline{a} \in \psi} \psi|_{a=true}) \rightarrow a$$

where $\psi|_{a=true}$ is the clause ψ where we have deleted $\overline{a}$. The purpose of $\psi|_{a=true}$ is to model what has to be true so that we can safely assign a to true, its more preferred value.

To return to our flying example, the hard constraints forbid b and $\overline{a}$ to be simultaneously true. This can be written as the clause $a \vee \overline{b}$. Hence, we have the constrained CP-net $\langle N, C\rangle$ where $N = \{a : b \succ \overline{b}, \overline{a} : \overline{b} \succ b, \overline{b} : a \succ \overline{a}. b : \overline{a} \succ a\}$ and $C = \{a \vee \overline{b}\}$. The optimality constraints corresponding to the given constrained CP-net are therefore $a \vee \overline{b}$ plus the following clauses:

$$(a \wedge a) \rightarrow b \qquad (b \wedge \overline{b}) \rightarrow \overline{a}$$
$$\overline{a} \rightarrow \overline{b} \qquad \overline{b} \rightarrow a$$

The only satisfying assignment for these constraints is ab. This is also the only optimal outcome in the constrained CP-net. In general, the satisfying assignments of the opti-

mality constraints are exactly the feasible and undominated outcomes of the constrained CP-net.

Theorem 2. *Given a constrained CP-net $\langle N, C\rangle$ over Boolean variables, an outcome is optimal for $\langle N, C\rangle$ iff it is a satisfying assignment of the optimality constraints $N \oplus_b C$.*

Proof. ($\Rightarrow$) Consider any outcome O that is optimal. Suppose that O does not satisfy $N \oplus C$. Clearly O satisfies C, since to be optimal it must be feasible (and undominated). Therefore O must not satisfy some $opt_C(p)$ where $p \in N$. The only way an implication is not satisfied is when the hypothesis is *true* and the conclusion is *false*. That is, $O \vdash \varphi$, $O \vdash \psi|_{a=true}$ and $O \vdash \overline{a}$ where $p = \varphi : a \succ \overline{a}$. In this situation, flipping from $\overline{a}$ to a would give us a new outcome O' such that $O' \vdash a$ and this would be an improvement according to p. However, by doing so, we have to make sure that the clauses in C containing $\overline{a}$ may now not be satisfied, since now $\overline{a}$ is false. However, we also have that $O \vdash \psi|_{a=true}$, meaning that if $\overline{a}$ is false these clauses are satisfied. Hence, there is an improving flip to another feasible outcome O'. But O was supposed to be undominated. Thus it is not possible that O does not satisfy $N \oplus_b C$. Therefore O satisfies all $opt_C(p)$ where $p \in N$. Since it is also feasible, O is a satisfying assignment of $N \oplus_b C$.

($\Leftarrow$) Consider any assignment O which satisfies $N \oplus_b C$. Clearly it is feasible as $N \oplus_b C$ includes C. Suppose we perform an improving flip in O. Without loss of generality, consider the improving flip from $\overline{a}$ to a. There are two cases. Suppose that this new outcome is not feasible. Then this new outcome does not dominate the old one in our semantics. Thus O is optimal. Suppose, on the other hand, that this new outcome is feasible. If this is an improving flip, there must exist a statement $\varphi : a \succ \overline{a}$ in N such that $O \vdash \varphi$. By assumption, O is a satisfying assignment of $N \oplus_b C$. Therefore $O \vdash opt(\varphi : a \succ \overline{a})$. Since $O \vdash \varphi$ and $O \vdash \overline{a}$, and *true* is not allowed to imply *false*, at least one $\psi|_{a=true}$ is not implied by O where $\psi \in C$ and $\overline{a} \in \psi$. However, as the new outcome is feasible, ψ has to be satisfied independent of how we set a. Hence, $O \vdash \psi|_{a=true}$. As this is a contradiction, this cannot be an improving flip. The satisfying assignment is therefore feasible and undominated. □

It immediately follows that we can test for feasible and undominated outcomes in linear time in the size of $\langle N, C\rangle$: we just need to test the satisfiability of the optimality constraints, which are as many as the constraints in C and the conditional statements in N. Notice that this construction works also for regular CP-nets without any hard constraints. In this case, the optimality constraints are of the form $\varphi \rightarrow a$ for each conditional preference statement $\varphi : a \succ \overline{a}$.

It was already known that optimality testing in acyclic CP-nets is linear [6]. However, our construction also works with cyclic CP-nets. Therefore optimality testing for cyclic CP-nets has now become an easy problem, even if the CP-nets are not constrained. On the other hand, determining if a constrained CP-net has any feasible and undominated outcomes is NP-complete (to show completeness, we map any SAT problem directly onto a constrained CP-net with no preferences). Notice that this holds also for acyclic CP-nets, and finding an optimal outcome in an acyclic constrained CP-net is NP-hard.

6 Non-Boolean variables

The construction in the previous section can be extended to handle variables whose domain contains more than 2 values. Notice that in this case the constraints are no longer clauses but regular hard constraints over a set of variables with a certain domain. Given a constrained CP-net $\langle N, C\rangle$, consider any conditional preference statement p for feature x in N of the form $\varphi : a_1 \succ a_2 \succ a_3$. For simplicity, we consider just 3 values. However, all the constructions and arguments extend easily to more values. The optimality constraints corresponding to this preference statement (let us call them $opt_C(p)$) are:

$$\varphi \wedge (C_x \wedge x = a_1) \downarrow_{var(C_x)-\{x\}} \rightarrow x = a_1$$

$$\varphi \wedge (C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}} \rightarrow x = a_1 \vee x = a_2$$

where C_x is the subset of constraints in C which involve variable x [2], and $\downarrow X$ projects onto the variables in X. The optimality constraints corresponding to $\langle N, C\rangle$ are again $N \oplus C = C \cup \{opt_C(p) \mid p \in N\}$. We can again show that this construction gives a new problem whose solutions are all the optimal outcomes of the constrained CP-net.

Theorem 3. *Given a constrained CP-net $\langle N, C\rangle$, an outcome is optimal for $\langle N, C\rangle$ iff it is a satisfying assignment of the optimality constraints $N \oplus C$.*

Proof. ($\Rightarrow$) Consider any outcome O that is optimal. Suppose that O does not satisfy $N \oplus C$. Clearly O satisfies C, since to be optimal it must be feasible (and undominated). Therefore O must not satisfy some $opt_C(p)$ where p preference statement in N. Without loss of generality, let us consider the optimality constraints $\varphi \wedge (C_x \wedge x = a_1) \downarrow_{var(C_x)-\{x\}} \rightarrow x = a_1$ and $\varphi \wedge (C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}} \rightarrow x = a_1 \vee x = a_2$ corresponding to the preference statement $\varphi : a_1 \succ a_2 \succ a_3$. The only way an implication is not satisfied is when the hypothesis is *true* and the conclusion is *false*. Let us take the first implication: $O \vdash \varphi$, $O \vdash (C_x \wedge x = a_1) \downarrow_{var(C_x)-\{x\}}$ and $O \vdash (x = a_2 \vee x = a_3)$. In this situation, flipping from $(x = a_2 \vee x = a_3)$ to $x = a_1$ would give us a new outcome O' such that $O' \vdash x = a_1$ and this would be an improvement according to p. However, by doing so, we have to make sure that the constraints in C containing $x = a_2$ or $x = a_3$ may now not be satisfied, since now $(x = a_2 \vee x = a_3)$ is false. However, we also have that $O \vdash (C_x \wedge x = a_1) \downarrow_{var(C_x)-\{x\}}$, meaning that if $x = a_1$ these constraints are satisfied. Hence, there is an improving flip to another feasible outcome O'. But O was supposed to be undominated. Therefore O satisfies the first of the two implications above.

Let us now consider the second implication: $O \vdash \varphi$, $O \vdash (C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}}$ and $O \vdash x = a_3$. In this situation, flipping from $x = a_3$ to $x = a_2$ would give us a new outcome O' such that $O' \vdash x = a_2$ and this would be an improvement according to p. However, by doing so, we have to make sure that the constraints in C containing $x = a_3$ may now not be satisfied, since now $x = a_3$ is false. However, we also have that $O \vdash (C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}}$, meaning that if $x = a_2$ these constraints are satisfied.

[2] More precisely, $C_x = \{c \in C | x \in con_c\}$.

Hence, there is an improving flip to another feasible outcome O'. But O was supposed to be undominated. Therefore O satisfies the second implication above. Thus O must satisfy all constraints $opt_C(p)$ where $p \in N$. Since it is also feasible, O is a satisfying assignment of $N \oplus C$.

($\Leftarrow$) Consider any assignment O which satisfies $N \oplus C$. Clearly it is feasible as $N \oplus C$ includes C. Suppose we perform an improving flip in O. There are two cases. Suppose that the outcomes obtained by performing any improving flip are not feasible. Then such new outcomes do not dominate the old one in our semantics. Thus O is optimal.

Suppose, on the other hand, that there is at least one new outcome, obtained via an improving flip, which is feasible. Assume the flips passes from $x = a_3$ to $x = a_2$. If this is an improving flip, without loss of generality, there must exist a statement $\varphi : \ldots \succ x = a_2 \succ x = a_3 \succ \ldots$ in N such that $O \vdash \varphi$. By hypothesis, O is a satisfying assignment of $N \oplus C$. Therefore $O \vdash opt(\varphi : \ldots \succ x = a_2 \succ \ldots \succ x = a_3 \succ \ldots)$ $= \varphi \wedge (C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}} \rightarrow \ldots \vee x = a_2$. Since $O \vdash \varphi$ and $O \vdash x = a_3$, and *true* is not allowed to imply *false*, O cannot satisfy $(C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}}$. But, as the new outcome, which contains $x = a_2$, is feasible, such constraints have to be satisfied independent of how we set x. Hence, $O \vdash (C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}}$. As this is a contradiction, this cannot be an improving flip to a feasible outcome. The satisfying assignment is therefore feasible and undominated. $\square$

Notice that the construction $N \oplus C$ for variables with more than two values in their domains is a generalization of the one for Boolean variables. That is, $N \oplus C = N \oplus_b C$ if N and C are over Boolean variables. Similar complexity results hold also now. However, while for Boolean variables one constraint is generated for each preference statement, now we generate as many constraints as the size of the domain minus 1. Therefore the optimality constraints corresponding to a constrained CP-net $\langle N, C \rangle$ are $|C| + | N | \times | D |$, where D is the domain of the variables. Testing optimality is still linear in the size of $\langle N, C \rangle$, if we assume D bounded. Finding an optimal outcome as usual requires us to find a solution of the constraints in $N \oplus C$, which is NP-hard in the size of $\langle N, C \rangle$.

7 CP-nets and soft constraints

It may be that we have soft and not hard constraints to add to our CP-net. For example, we may have soft constraints representing other quantitative preferences. In the rest of this section, a constrained CP-net will be a pair $\langle N, C \rangle$, where N is a CP-net and C is a set of soft constraints. Notice that this definition generalizes the one given in Section 6 since hard constraints can be seen as a special case of soft constraints (see Section 2).

The construction of the optimality constraints for constrained CP-nets can be adapted to work with soft constraints. To be as general as possible, variables can again have more than two values in their domains. The constraints we obtain are very similar to those of the previous sections, except that now we have to reason about optimization as soft constraints define an optimization problem rather than a satisfaction problem.

Consider any CP statement p of the form $\varphi : x = a_1 \succ x = a_2 \succ x = a_3$. For simplicity, we again consider just 3 values. However, all the constructions and ar-

guments extend easily to more values. The optimality constraints corresponding to p, called $opt_{soft}(p)$, are the following hard constraints:

$$\varphi \wedge cut_{best(C)}((\varphi \wedge C_x \wedge x = a_1) \downarrow_{var(C_x)-\{x\}}) \rightarrow x = a_1$$

$$\varphi \wedge cut_{best(C)}((\varphi \wedge C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}}) \rightarrow x = a_1 or x = a_2$$

where C_x is the subset of soft constraints in C which involve variable x, $best(S)$ is the highest preference value for a complete assignment of the variables in the set of soft constraints S, and $cut_\alpha S$ is a hard constraint obtained from the soft constraint S by forbidding all tuples which have preference value less than α in S. The optimality constraints corresponding to $\langle N, C\rangle$ are $C_{opt}(\langle N, C\rangle) = \{opt_{soft}(p) \mid p \in N\}$.

Consider a CP-net with two features, X and Y, such that the domain of Y contains y_1 and y_2, while the domain of X contains x_1, x_2, and x_3. Moreover, we have the following CP-net preference statements: $y_1 \succ y_2$, $y_1 : x_1 \succ x_2 \succ x_3$, $y_2 : x_2 \succ x_1 \succ x_3$. We also have a soft (fuzzy) unary constraint over X, which gives the following preferences over the domain of X: 0.1 to x_1, 0.9 to x_2, and 0.5 to x_3. By looking at the CP-net alone, the ordering over the outcomes is given by $y_1x_1 \succ y_1x_2 \succ y_1x_3 \succ y_2x_3$ and $y_1x_2 \succ y_2x_2 \succ y_2x_1 \succ y_2x_3$. Thus y_1x_1 is the only optimal outcome of the CP-net. On the other hand, by taking the soft constraint alone, the optimal outcomes are all those with $X = x_2$ (thus y_1x_2 and y_2x_2).

Let us now consider the CP-net and the soft constraints together. To generate the optimality constraints, we first compute best(C), which is 0.9. Then, we have:

- for statement $y_1 \succ y_2$: $Y = y_1$;
- for statement $y_1 : x_1 \succ x_2 \succ x_3$: we generate the constraints $Y = y_1 \wedge false \rightarrow X = x_1$ and $Y = y_1 \wedge Y = y_1 \rightarrow X = x_1 \vee X = x_2$. Notice that we have false in the condition of the first implication because $cut_{0.9}(Y = y_1 \wedge C_x \wedge X = x_1) \downarrow_Y = false$. On the other hand, in the condition of the second implication we have $cut_{0.9}(Y = y_1 \wedge C_x \wedge X = x_2) \downarrow_Y = (Y = y_1)$. Thus, by removing false, we have just one constraint: $Y = y_1 \rightarrow X = x_1 \vee X = x_2$;
- for statement $y_2 : x_2 \succ x_1 \succ x_3$: similarly to above, we have the constraint $Y = y_2 \rightarrow X = x_2$.

Let us now compute the optimal solutions of the soft constraint over X which are also feasible for the following set of constraints: $Y = y_1, Y = y_1 \rightarrow X = x_1 \vee X = x_2$, $Y = y_2 \rightarrow X = x_2$. The only solution which is optimal for the soft constraints and feasible for the optimality constraints is y_1x_2. Thus this solution is optimal for the constrained CP-net.

Notice that the optimal outcome for the constrained CP-net of the above example is not optimal for the CP-net alone. In general, an optimal outcome for a constrained CP-net has to be optimal for the soft constraints, and such that there is no other outcome which can be reached from it in the ordering of the CP-net with an improving chain of optimal outcomes. Thus, in the case of CP-nets constrained by soft constraints, Definition 2 is replaced by the following one:

Definition 3 **($O1 \succ_{soft} O_2$).** *Given a constrained CP-net $\langle N, C\rangle$, where C is a set of soft constraints, outcome O_1 is* **better** *than outcome O_2 (written $O_1 \succ_{soft} O_2$) iff there*

is a chain of flips from O_1 to O_2, where each flip is worsening for N and each outcome in the chain is optimal for C.

Notice that this definition is just a generalization of Def. 2, since optimality in hard constraints is simply feasibility. Thus $\succ = \succ_{soft}$ when C is a set of hard constraints.

Consider the same CP-net as in the previous example, and a binary fuzzy constraint over X and Y which gives preference 0.9 to $x_2 y_1$ and $x_1 y_2$, and preference 0.1 to all other pairs. According to the above definition, both $x_2 y_1$ and $x_1 y_2$ are optimal for the constrained CP-net, since they are optimal for the soft constraints and there are no improving path of optimal outcomes between them in the CP-net ordering. Let us check that the construction of the optimality constraints obtains the same result:

- for $y_1 \succ y_2$ we get $cut_{0.9}(C_y \wedge Y = y_1) \downarrow_X \rightarrow Y = y_1$. Since $cut_{0.9}(C_y \wedge Y = y_1) \downarrow_X = (X = x_2)$, we get $X = x_2 \rightarrow Y = y_1$.
- for statement $y_1 : x_1 \succ x_2 \succ x_3$: $Y = y_1 \wedge cut_{0.9}(Y = y_1 \wedge C_x \wedge X = x_1) \downarrow_Y \rightarrow X = x_1$. Since $cut_{0.9}(Y = y_1 \wedge C_x \wedge X = x_1) \downarrow_Y = false$, we get a constraint which is always true. Also, we have the constraint $Y = y_1 \wedge cut_{0.9}(Y = y_1 \wedge C_x \wedge X = x_2) \downarrow_Y \rightarrow X = x_1 \vee X = x_2$. Since $cut_{0.9}(Y = y_1 \wedge C_x \wedge X = x_2) \downarrow_Y = (Y - y_1)$, we get $Y = y_1 \wedge Y = y_1 \rightarrow X = x_1 \vee X = x_2$.
- for statement $y_2 : x_2 \succ x_1 \succ x_3$: similarly to above, we have the constraint $Y = y_2 \rightarrow X = x_2 \vee X = x_1$.

Thus the set of optimality constraints is the following one: $X = x_2 \rightarrow Y = y_1$, $Y = y_1 \rightarrow X = x_1 \vee X = x_2$, and $Y = y_2 \rightarrow X = x_2 \vee X = x_1$. The feasible solutions of this set of constraints are $x_2 y_1$, $x_1 y_1$, and $x_1 y_2$. Of these constraints, the optimal outcomes for the soft constraint are $x_2 y_1$ and $x_1 y_2$. Notice that, in the ordering induced by the CP-net over the outcomes, these two outcomes are not linked by a path of improving flips through optimal outcomes for the soft constraints. Thus they are both optimal for the constrained CP-net.

Theorem 4. *Given a constrained CP-net $\langle N, C\rangle$, where C is a set of soft constraints, an outcome is optimal for $\langle N, C\rangle$ iff it is an optimal assignment for C and if it satisfies $C_{opt}(\langle N, C\rangle)$.*

Proof. ($\Rightarrow$) Consider an outcome O that is optimal for $\langle N, C\rangle$. Then by definition it must be optimal for C. Suppose the outcome does not satisfy C_{opt}. Therefore O must not satisfy some constraint $opt_C(p)$ where p preference statement in N. Without loss of generality, let us consider the optimality constraints

$$\varphi \wedge cut_{best(C)}((\varphi \wedge C_x \wedge x = a_1) \downarrow_{var(C_x)-\{x\}}) \rightarrow x = a_1$$

$$\varphi \wedge cut_{best(C)}((\varphi \wedge C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}}) \rightarrow x = a_1 or x = a_2$$

corresponding to the preference statement $\varphi : a_1 \succ a_2 \succ a_3$.

The only way an implication is not satisfied is when the hypothesis is *true* and the conclusion is *false*. Let us take the first implication: $O \vdash \varphi$, $O \vdash cut_{best(C)}((\varphi \wedge C_x \wedge x = a_1) \downarrow_{var(C_x)-\{x\}})$ and $O \vdash (x = a_2 \vee x = a_3)$. In this situation, flipping from $(x = a_2 \vee x = a_3)$ to $x = a_1$ would give us a new outcome O' such that

$O' \vdash x = a_1$ and this would be an improvement according to p. However, by doing so, we have to make sure that the soft constraints in C containing $x = a_2$ or $x = a_3$ may now still be satisfied optimally, since now $(x = a_2 \vee x = a_3)$ is false. We also have that $O \vdash cut_{best(C)}((\varphi \wedge C_x \wedge x = a_1) \downarrow_{var(C_x)-\{x\}})$, meaning that if $x = a_1$ these constraints are satisfied optimally. Hence, there is an improving flip to another outcome O' which is optimal for C and which satisfies C_{opt}. But O was supposed to be undominated. Therefore O satisfies the first of the two implications above.

Let us now consider the second implication: $O \vdash \varphi$, $O \vdash cut_{best(C)}((\varphi \wedge C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}})$, and $O \vdash x = a_3$. In this situation, flipping from $x = a_3$ to $x = a_2$ would give us a new outcome O' such that $O' \vdash x = a_2$ and this would be an improvement according to p. However, by doing so, we have to make sure that the constraints in C containing $x = a_3$ may now still be satisfied optimally, since now $x = a_3$ is false. However, we also have that $O \vdash cut_{best(C)}((\varphi \wedge C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}})$, meaning that if $x = a_2$ these constraints are satisfied optimally. Hence, there is an improving flip to another feasible outcome O'. But O was supposed to be undominated. Therefore O satisfies the second implication above. Thus O must satisfy all the optimality constraints $opt_C(p)$ where $p \in N$.

($\Leftarrow$) Consider any assignment O which is optimal for C and satisfies C_{opt}. Suppose we perform a flip on O. There are two cases. Suppose that the new outcome is not optimal for C. Then the new outcome does not dominate the old one in our semantics. Thus O is optimal. Suppose, on the other hand, that there is at least one new outcome, obtained via an improving flip, which is optimal for C and satisfies C_{opt}. Assume the flip passes from $x = a_3$ to $x = a_2$. If this is an improving flip, without loss of generality, there must exist a statement $\varphi : \ldots \succ x = a_2 \succ x = a_3 \succ \ldots$ in N such that $O \vdash \varphi$. By hypothesis, O is an optimal assignment of C and satisfies C_{opt}. Therefore $O \vdash opt(\varphi : \ldots \succ x = a_2 \succ \ldots \succ x = a_3 \succ \ldots) = \varphi \wedge cut_{best(C)}((\varphi \wedge C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}}) \rightarrow \ldots \vee x = a_2$.

Since $O \vdash \varphi$ and $O \vdash x = a_3$, and *true* is not allowed to imply *false*, O cannot satisfy $cut_{best(C)}((\varphi \wedge C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}})$. But O', which contains $x = a_2$, is assumed to be optimal for C, so $cut_{best(C)}((\varphi \wedge C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}})$ has to be satisfied independently of how we set x. Hence, $O \vdash (C_x \wedge x = a_2) \downarrow_{var(C_x)-\{x\}}$. As this is a contradiction, this cannot be an improving flip to an outcome which is optimal for C and satisfies C_{opt}. Thus O is optimal for the constrained CP-net. $\square$

It is easy to see how the construction of this section can be used when a CP-net is constrained by a set of both hard and soft constraints, or by several sets of hard and soft constraints, since they can all be seen as just one set of soft constraints.

Let us now consider the complexity of constructing the optimality constraints and of testing or finding optimal outcomes, in the case of CP-nets constrained by soft constraints. First, as with hard constraints, the number of optimality constraints we generate is $|N|| \times (|D| - 1)$, where $|N|$ is the number of preference statements in N and D is the domain of the variables. Thus we have $|C_{opt}(\langle N, C\rangle)| = |N| \times (|D| - 1)$. To test if an outcome O is optimal, we need to check if O satisfies C_{opt} and if it is optimal for C. Checking feasibility for C_{opt} takes linear time in $|N| \times (|D| - 1)$. Then, we need to check if O is optimal for C. This is NP-hard the first time we do it, otherwise (if the optimal preference value for C is known) is linear in the size of C. To find an optimal

outcome, we need to find the optimals for C which are also feasible for C_{opt}. Finding optimals for C needs exponential time in the size of C, and checking feasibility in C_{opt} is linear in the size of C_{opt}. Thus, with respect to the corresponding results for hard constraints, we only need to do more work the first time we want to test an outcome for optimality.

8 Multiple constrained CP-nets

There are situations when we need to represent the preferences of multiple agents. For example, when we are scheduling workers, each will have a set of preferences concerning the shifts. These ideas generalize to such a situation. Consider several CP-nets N_1, $\ldots$, N_k, and a set of hard or soft constraints C. We will assume for now that all the CP nets have the same features. To begin, we will say an outcome is optimal iff it is optimal for each constrained CP net $\langle N_i, C\rangle$. This is a specific choice but we will see later that other choices can be considered as well. We will call this notion of optimality, *All-optimal*.

Definition 4 (All-optimal). *Given a multiple constrained CP net $M = \langle(N_1,\ldots, N_k), C\rangle$, an outcome O is All-optimal for M if O if it is optimal for each constrained CP net $\langle N_i, C\rangle$.*

This definition, together with Theorem 4, implies that to find the all-optimal outcomes for M we just need to generate the optimality constraints for each constrained CP net $\langle N_i, C\rangle$, and then take the outcomes which are optimal for C and satisfy all optimality constraints.

Theorem 5. *Given a multiple constrained CP net $M = \langle(N_1, \ldots, N_k), C\rangle$, an outcome O is All-optimal for M iff O is optimal for C and it satisfies the optimality constraints in $\bigcup_i C_{opt}(\langle N_i, C\rangle)$.*

This semantics is one of consensus: all constrained CP nets must agree that an outcome is optimal to declare it optimal for the multiple constrained CP net. Choosing this semantics obviously satisfies all CP nets. However, there could be no outcome which is optimal. In [8] a similar consensus semantics (although for multiple CP nets, with no additional constraints) is called Pareto optimality, and it is one among several alternative to aggregate preferences expressed via several CP nets. This semantics, adapted to our context, would be defined as follows:

Definition 5 (Pareto). *Given a multiple constrained CP net $M = \langle(N_1, \ldots, N_k), C\rangle$, an outcome O is Pareto-better than an outcome O' iff it is better for each constrained CP net. It is Pareto-optimal for M iff there is no other outcome which is Pareto-better.*

If an outcome is all-optimal, it is also Pareto-optimal. However, the converse is not true in general. These two semantics may seem equally reasonable. However, while all-optimality can be computed via the approach of this paper, which avoids dominance testing, Pareto optimality needs such tests, and therefore it is in general much more

expensive to compute. In particular, whilst optimality testing for Pareto optimality requires exponential time, for All-optimality it just needs linear time (in the sum of the sizes of the CP nets).

Other possibilities proposed in [8] require optimals to be the best outcomes for a majority of CP nets (this is called Majority), or for the highest number of CP nets (called Max). Other semantics like Lex, associate each CP net with a priority, and then declare optimal those outcomes which are optimal for the CP nets with highest priority, in a lexicographical fashion. In principle, all these semantics can be adapted to work with multiple constrained CP nets. However, as for Pareto optimality, whilst their definition is possible, reasoning with them would require more than just satisfying a combination of the optimality constraints, and would involve dominance testing.

Thus the main gain from our semantics (all-optimal and others that can be computed via this approach) is that dominance testing is not required. This makes optimality testing (after the first test) linear rather than exponential, although finding optimals remains difficult (as it is when we find the optimals of the soft constraints and check the feasibility of the optimality constraints).

9 Related work

The closest work is [2], where acyclic CP nets are constrained via hard constraints, and an algorithm is proposed to find one or all the optimal outcomes of the constrained CP net. However, there are several differences. First, the notion of optimality in this previous approach is different from the one used here: in [2], an outcome O is optimal if satisfies the constraints and there is no other feasible outcome which is better than it in the CP net ordering. Therefore, if two outcomes are both feasible and there is an improving path from one to the other one in the CP net, but they are not linked by a path of feasible outcomes, then in this previous approach only the highest one is optimal, while in ours they are both optimal. For example, assume we have a CP net with two Boolean features, A and B, and the following CP statements: $a \succ \overline{a}$, $a : b \succ \overline{b}$, $\overline{a} : \overline{b} \succ b$, and the constraint $\overline{a} \vee b$ which rules out $a\overline{b}$. Then, the CP net ordering on outcomes is $ab \succ a\overline{b} \succ \overline{a}\overline{b} \succ \overline{a}b$. In our approach, both ab and $\overline{a}\overline{b}$ are optimal, whilst in in the previous approach only ab is optimal. Thus we obtain a superset of the optimals computed in the previous approach.

Reasoning about this superset is, however, computationally more attractive. To find the first optimal outcome, the algorithm in [2] uses branch and bound and thus has a complexity that is comparable to solving the set of constraints. Then, to find other optimal outcomes, they need to perform dominance tests (as many as the number of optimal outcomes already computed), which are very expensive. In our approach, to find one optimal outcome we just need to solve a set of optimality constraints, which is NP-hard.

Two issues that are not addressed in [2] are testing optimality efficiently and reasoning with cyclic CP nets. To test optimality, we must run the branch and bound algorithm to find all optimals, and stop when the given outcome is generated or when all optimals are found. In our approach, we check the feasibility of the given outcome with respect to the optimality constraints. Thus it takes linear time. Our approach is based on the CP

statements and not on the topology of the dependency graph. Thus it works just as well with cyclic CP nets.

Another related work is [7], where CP nets orderings are approximated via a set of soft constraints. The approximation here is not needed, since we are not trying to model the entire ordering over outcomes, but only the set of optimals.

Finally, our construction can be seen as a generalization of that given in Section 4 of [4], where they treat the case of mapping a CP net on Boolean features, without any constraints, onto a SAT problem.

10 Conclusions

We have presented a novel approach to deal with preferences expressed as a mixture of hard constraints, soft constraints, and CP nets. The main idea is to generate a set of hard constraints whose solutions are optimal for the preferences. Our approach focuses on finding and testing optimal solutions. It avoids the costly dominance tests previously used to reason about CP nets. To represent the preferences of multiple agents, we have also considered multiple CP nets. We have shown that it is possible to define semantics for preference aggregation for multiple CP nets which also avoid dominance testing. One of the main advantages of this simple and elegant technique is that it permits conventional constraint and SAT solvers to solve problems involving both preferences and constraints.

Acknowledgements. This work is partially supported by ASI (Italian Space Agency) under project ARISCOM (Contract I/R/215/02).

References

1. S. Bistarelli, U. Montanari, F. Rossi. Semiring-based Constraint Solving and Optimization. Journal of ACM, vol. 44, n. 2, pp. 201-236, March 1997.
2. C. Boutilier, R. Brafman, C. Domshlak, H. Hoos and D. Poole. Preference-based constraint optimization with CP-nets. Computational Intelligence, vol. 20, n. 2, pp. 137-157, May 2004.
3. C. Boutilier, R. Brafman, H. Hoos, and D. Poole. Reasoning with conditional ceteris paribus preference statements. In *Proceedings of 15th Annual Conference on Uncertainty in Artificial Intelligence (UAI-99)*, pp. 71-80, Stockholm, Sweden, 1999.
4. R. Brafman and Y. Dimopoulos. Extended Semantics and Optimization Algorithms for CP-Networks. Computational Intelligence, vol. 20, n. 2, pp. 218-245, May 2004.
5. R. Dechter. Constraint processing. Morgan Kaufmann, 2003.
6. C. Domshlak, R. I. Brafman. CP-nets - reasoning and consistency testing. In *Proceedings of the Eighth International Conference on Principles of Knowledge Representation and Reasoning (KR-02)*, pages 121–132. Morgan Kaufmann, Toulose, France, 2002.
7. C. Domshlak, F. Rossi, K. B. Venable, T. Walsh. Reasoning about soft constraints and conditional preferences: complexity results and approximation techniques. In *Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence (IJCAI-03)*, Morgan Kaufmann, Acapulco, Mexico, August 2003.
8. F. Rossi, K. B. Venable, T. Walsh. mCP nets: representing and reasoning with preferences of multiple agents. In *Proceeding of the Nineteenth National Conference on Artificial Intelligence (AAAI-04)*, San Jose, CA, USA, July 2004.

Efficient Evaluation of Disjunctive Datalog Queries with Aggregate Functions

Manuela Citrigno[1], Wolfgang Faber[2], Gianluigi Greco[3], and Nicola Leone[4]

[1] DEIS, Università di Bologna, 40136 Bologna, Italy
citrigno@deis.unibo.it
[2] Institut für Informationssysteme, TU Wien, 1040 Wien, Austria
faber@kr.tuwien.ac.at
[3] DEIS, Università della Calabria, 87030 Rende, Italy
ggreco@si.deis.unical.it
[4] Dip. di Matematica, Università della Calabria, 87030 Rende, Italy
leone@mat.unical.it

Abstract. We present a technique for the optimization of (partially) bound queries over disjunctive datalog programs enriched with aggregate functions (Datalog$^{\vee\mathcal{A}}$ programs). This class of programs has been recently proved to be well-suited for declaratively formalizing repair semantics in data integration systems. Indeed, even though disjunctive programs provide a natural way for encoding the possible repairs (i.e., insertions or deletions of tuples) of an inconsistent database, they do not suffice for applications in real scenarios, where users usually want to build summary views of data residing in different databases.
The technique exploits the propagation of query bindings, and extends the Magic-Set optimization technique (originally defined for non-disjunctive programs without aggregate functions) to Datalog$^{\vee\mathcal{A}}$ programs. All the algorithms presented in the paper have been fully integrated and implemented in the DLV system – the state-of-the-art implementation of disjunctive datalog.

1 Introduction

Disjunctive datalog (Datalog$^{\vee}$) programs are logic programs where disjunction may occur in the heads of rules [12, 11]. Disjunctive datalog is very expressive in a precise mathematical sense: it allows to express every property of finite ordered structures that is decidable in the complexity class Σ_2^P ($\mathrm{NP}^{\mathrm{NP}}$) [11]. Therefore, under widely believed assumptions, Datalog$^{\vee}$ is strictly more expressive than *normal* (*disjunction-free*) datalog which can express only problems of lower complexity. Importantly, besides enlarging the class of applications which can be encoded in the language, disjunction often allows for representing problems of lower complexity in a simpler and more natural fashion [10].

Recently, disjunctive datalog is employed in several "hot" application areas like information integration and knowledge management. In particular, several

approaches formalizing repair semantics in data integration system by using logic programs have been proposed (see, e.g., [1, 13, 7]), and the exploitation of disjunctive datalog for information integration is the main focus of the INFOMIX project (IST-2001-33570), funded by the European Commission.

Data integration is an important problem, given that more and more data are dispersed over many data sources. In a user-friendly information system, a data integration system provides transparent access to the data, and relieves the user from the burden of having to identify the relevant data sources for a query, accessing each of them separately, and combining the individual results into the global view of the data.

Informally, a data integration system $\mathcal{I}$ may be viewed as system $\langle \mathcal{G}, \mathcal{S}, \mathcal{M} \rangle$ that consists of a global schema $\mathcal{G}$, which specifies the global (user) elements, a source schema $\mathcal{S}$, which describes the structure of the data sources in the system, and a mapping $\mathcal{M}$, which specifies the relationship between the sources and the global schema. Usually, the global schema also contains information about constraints, Σ, such as key constraints or exclusion dependencies issued on a relational global schema. When the user issues a query q on the global schema, the global database is constructed by data retrieval from the sources and q is answered from it. However, the global database might be inconsistent with the constraints Σ.

To remedy this problem, the inconsistency might be eliminated by modifying the database and reasoning on the "repaired" database. To this aim, the idea exploited in the mentioned papers is to encode the constraints Σ of $\mathcal{G}$ into a logic program, Π, using disjunction (or unstratified negation, as done in [7]), such that the stable models of this program yield the repairs of the global database. Answering a user query, q, then amounts to cautious reasoning over the logic program Π augmented with the query, and the retrieved facts $\mathcal{R}$.

An attractive feature of this approach is that disjunctive logic programs serve as executable logical specifications of repair, and thus allow to state repair policies in a declarative manner rather than in a procedural way. Moreover, the effectiveness of the approach is guaranteed by the availability of some efficient inference engines, such as the DLV system [19] and the GnT system [17], and by some optimization techniques for disjunctive programs which have been recently proposed in [14, 8].

However, in spite of its high expressiveness, it has been clearly recognized that classical Datalog$^\vee$ presents some limitations for its applicability to real data integration settings. In data integration contexts, it is reasonable to assume that users should be able to express most of the SQL2 queries. Indeed, SQL2 is widely accepted as the standard query language in the database context. However, Datalog$^\vee$ is not comparable with SQL2 in that some queries that can be expressed in Datalog$^\vee$ cannot be expressed in SQL2 and vice versa. For instance, Datalog$^\vee$ provides the power of recursion and disjunction which cannot be simulated in SQL2, and SQL2 allows aggregate operators (such as SUM, MIN, MAX) and ordering features which cannot be expressed or easily simulated

in classical Datalog$^{\vee}$. In some cases, aggregate operators can be simulated in Datalog$^{\vee}$, but this produces inefficient programs and unnatural encodings of the problems.

In [9], the above deficiency of Datalog$^{\vee}$ has been overcome by extending the language with a sort of aggregate functions (Datalog$^{\vee\mathcal{A}}$), first studied in the context of deductive databases, and implementing them in DLV [10] – the state-of-the-art Disjunctive Logic Programming system. Under a computational point of view, the resulting formalism turned out to be equivalent to standard Datalog$^{\vee}$, since 'brave reasoning' for ground programs is Σ_2^P-complete whereas 'cautious reasoning' for ground programs is Π_2^P-complete. However, at the best of our knowledge, no optimizations techniques for the efficient evaluation of disjunctive programs enriched with aggregate functions have been appeared in the literature. Hence, with current implementations of stable model engines, the evaluation of queries over large data sets quickly becomes infeasible because of lacking scalability. This calls for suitable optimization methods that help in speeding up the evaluation of queries, and in making Datalog$^{\vee\mathcal{A}}$ well suited for real applications in data integration settings.

In this paper, we face such efficiency problems and we present an optimization technique, that is able to support Datalog$^{\vee}$ programs, enriched with aggregate functions. Specifically, we investigate a promising line of research consisting of the extension of deductive database techniques and, specifically, of binding propagation techniques exploited in the Magic-Set method [24, 2, 4, 23, 18, 22], to nonmonotonic logic languages like disjunctive datalog.

1.1 Related Work

The Magic-Set method is one of the most well-known technique for the optimization of positive recursive Datalog programs due to its efficiency and its generality, even though other focused methods such as the supplementary magic set and other special techniques for linear and chain queries have been proposed as well (see, e.g., [15, 24, 21]). Intuitively, the goal of the Magic-Set method (originally defined for non-disjunctive datalog queries only) is to use the constants appearing in the query to reduce the size of the instantiation by eliminating "a priori" a number of ground instances of the rules which cannot contribute to the derivation of the query goal.

After seminal papers [2, 4], the viability of the approach was demonstrated e.g. in [16, 20]. Lateron, extensions and refinements have been proposed, addressing e.g. query constraints in [23], the well-founded semantics in [18], or integration into cost-based query optimization in [22]. The research on variations of the Magic-Set method is still going on. For instance, in [5] a technique for the class of *soft-stratifiable* programs is given, and in [14] an elaborated technique for disjunctive programs is described.

It has been noted (e.g. in [18]) that in the non-disjunctive case, memoing techniques lead to similar computations as evaluations after Magic-Set transformations. Also in the disjunctive case such techniques have been proposed, e.g.

Hyper Tableaux [3], for which similar relations might hold. However, we leave this issue for future research, and follow [18] in noting that an advantage of Magic-Sets over such methods is that the latter may be more easily combined with other database optimization techniques.

An extension of the Magic-Set method to disjunctive programs is due to [14], where the author observes that binding propagation strategies have to be changed for disjunctive rules so that each time a head predicate receives some binding from the query, it eventually propagates this relevant information to all the other head predicates as well as to the body predicates. An algorithm implementing the above strategy has been also proposed in [14]. Moreover, in [8] some fresh and refined ideas for extending the Magic-Set method to disjunctive datalog queries have been provided, by avoiding some major drawbacks that are intrinsic of the method in [14].

1.2 Contribution

In this paper, we continue on the way paved in [8], and we provide an extension of the Magic-Set method to deal with Datalog$^{\vee\mathcal{A}}$ programs as well (DMS$^{\mathcal{A}}$ algorithm). Specifically, in Section 2, we preliminarily show how to extend Disjunctive Logic Programming by aggregate functions and we formally define the semantics of the resulting language, named Datalog$^{\vee\mathcal{A}}$.

Then, in Section 3, we show that in order to make such technique work in the presence of both disjunction and aggregate atoms, traditional Sideways Information Passing Strategies (*SIPS*), cf. [4], simulating the data flow occurring in the top-down evaluation of the query, must be modified by imposing some additional constraints. We provide all the details needed for understanding the main ideas exploited in the design of the DMS$^{\mathcal{A}}$ algorithm, which has been fully implemented and integrated in the DLV system [19] – the state-of-the-art implementation of disjunctive datalog. Finally, in Section 4 we draw our conclusions.

2 The Datalog$^{\vee\mathcal{A}}$ Language

In this section, we provide a formal definition of the syntax and semantics of the Datalog$^{\vee\mathcal{A}}$ language – an extension of Datalog$^{\vee}$ by set-oriented functions (also called aggregate functions). We assume that the reader is familiar with standard Datalog$^{\vee}$; we refer to atoms, literals, rules, and programs of Datalog$^{\vee}$, as *standard atoms, standard literals, standard rules*, and *standard programs,* respectively. For further background, see [12, 10].

2.1 Syntax

A (Datalog$^{\vee\mathcal{A}}$) *set* is either a symbolic set or a ground set. A *symbolic set* is a pair $\{Vars : Conj\}$, where *Vars* is a list of variables and *Conj* is a conjunction of

standard literals. Intuitively, a symbolic set $\{X{:}a(X,Y),p(Y)\}$ stands for the set of X-values making $a(X,Y),p(Y)$ true, i.e., $\{X : \exists Y s.t.\ a(X,Y),p(Y)\ \textit{is true}\}$. Note that also negative literals may occur in the conjunction $Conj$ of a symbolic set.

A *ground set* is a set of pairs of the form $\langle \bar{t} : Conj \rangle$, where $\bar{t}$ is a list of constants and $Conj$ is a ground (variable free) conjunction of standard literals. An *aggregate function* is of the form $f(S)$, where S is a set, and f is a *function name* among `#count`, `#min`, `#max`, `#sum`, `#times`. An *aggregate atom* is $Lg \prec_1 f(S) \prec_2 Rg$, where $f(S)$ is an aggregate function, $\prec_1, \prec_2 \in \{=, <, \leq, >, \geq\}$, and Lg and Rg (called *left guard*, and *right guard*, respectively) are terms. One of "$Lg \prec_1$" and "$\prec_2 Rg$" can be omitted. An *atom* is either a standard (Datalog$^{\vee}$) atom or an aggregate atom.

A *(Datalog$^{\vee\mathcal{A}}$) rule* r is a construct

$$\mathtt{a_1\ v\ \cdots\ v\ a_n\ {:}\text{-}\ b_1, \cdots, b_m.}$$

where $a_1, \cdots, a_n$, $b_1, \cdots, b_m$ are atoms, and $n \geq 0$, $m \geq 0$. The disjunction $a_1\ \mathtt{v}\ \cdots\ \mathtt{v}\ a_n$ is the *head* of r, while the conjunction $b_1, ..., b_m$ is the *body* of r. A *(Datalog$^{\vee\mathcal{A}}$) program* is a set of Datalog$^{\vee\mathcal{A}}$ rules.

For simplicity, and without loss of generality, we assume that the body of each rule contains at most one aggregate atom. A *global* variable of a rule r is a variable appearing in some standard atom of r; a *local* variable of r is a variable appearing solely in an aggregate function in r.

Stratification. A Datalog$^{\vee\mathcal{A}}$ program P is *aggregate-stratified* if there exists a function $||\ ||$, called *level mapping*, from the set of (standard) predicates of P to ordinals, such that for each pair a and b of (standard) predicates of P, and for each rule $r \in \mathcal{P}$: (i) if a appears in the head of r, and b appears in an aggregate atom in the body of r, then $||b|| < ||a||$, and (ii) if a appears in the head of r, and b occurs in a standard atom in the body of r, then $||b|| \leq ||a||$.

Example 1. Consider the program consisting of a set of facts for predicates a and b, plus the following two rules:

$$\mathtt{q(X)\ {:}\text{-}\ p(X), \#count\{Y : a(Y,X), b(X)\} \leq 2.}$$
$$\mathtt{p(X)\ {:}\text{-}\ q(X), b(X).}$$

The program is aggregate-stratified, as the following level mapping $||\ ||$ satisfies the required conditions: $||\mathtt{a}|| = ||\mathtt{b}|| = \mathtt{1}$; $||\mathtt{p}|| = ||\mathtt{q}|| = \mathtt{2}$.
If we add the rule $\mathtt{b(X)\ {:}\text{-}\ p(X)}$, then no legal level-mapping exists and the program becomes aggregate-unstratified. □

Intuitively, aggregate-stratification forbids recursion through aggregates, which could cause an unclear semantic in some cases. Consider, for instance, the (aggregate-unstratified) program consisting only of rule $\mathtt{p(a)\ {:}\text{-}\ \#count\{X : p(X)\} = 0}$. Neither $\mathtt{p(a)}$ nor $\emptyset$ is an intuitive meaning for the program. We should

probably assert that the above program does not have any answer set (defining a notion of "stability" for aggregates), but then positive programs would not always have an answer set if there is no integrity constraint. In the following we assume that Datalog$^{\vee\mathcal{A}}$ programs are safe and aggregate-stratified.

2.2 Semantics

Given a Datalog$^{\vee\mathcal{A}}$ program $\mathcal{P}$, let $U_{\mathcal{P}}$denote the set of constants appearing in $\mathcal{P}$, $U_{\mathcal{P}}^{\mathcal{N}} \subseteq U_{\mathcal{P}}$ the set of the natural numbers occurring in $U_{\mathcal{P}}$, and $B_{\mathcal{P}}$the set of standard atoms constructible from the (standard) predicates of $\mathcal{P}$ with constants in $U_{\mathcal{P}}$. Furthermore, given a set S, $\overline{2}^S$ denotes the set of all multisets over elements from S. Let us now describe the domains and the meanings of the aggregate functions we consider.
#`count`: defined over $\overline{2}^{U_{\mathcal{P}}}$, returns the number of the elements in the set.
#`sum`: defined over $\overline{2}^{U_{\mathcal{P}}^{\mathcal{N}}}$, returns the sum of the elements in the set.
#`times`: defined over $\overline{2}^{U_{\mathcal{P}}^{\mathcal{N}}}$, returns the product of the elements in the set.[5]
#`min` ; #`max`: defined over $\overline{2}^{U_{\mathcal{P}}} - \emptyset$, returns the minimum/maximum element in the set (if the set contains also strings, the lexicographic ordering is considered).
If the argument of an aggregate function does not belong to its domain, then $\bot$ is returned.

A *substitution* is a mapping from a set of variables to the set $U_{\mathcal{P}}$ of the constants appearing in the program $\mathcal{P}$. A substitution from the set of global variables of a rule r (to $U_{\mathcal{P}}$) is a *global substitution for r*; a substitution from the set of local variables of a symbolic set S (to $U_{\mathcal{P}}$) is a *local substitution for S*. Given a symbolic set without global variables $S = \{\mathit{Vars} : \mathit{Conj}\}$, the *instantiation of set S* is the following ground set of pairs *inst(S)*:
$\{\langle\gamma(\mathit{Vars}) : \gamma(\mathit{Conj})\rangle \mid \gamma \text{ \textit{is a local substitution for} } S\}$. Given a substitution σ and a Datalog$^{\vee\mathcal{A}}$ object Obj (rule, conjunction, set, etc.), with a little abuse of notation, we denote by $\sigma(\mathit{Obj})$ the object obtained by replacing each variable X in Obj by $\sigma(X)$.

A *ground instance* of a rule r is obtained in two steps: (1) a global substitution σ for r is first applied over r; (2) every symbolic set S in $\sigma(r)$ is replaced by its instantiation *inst(S)*. The instantiation $\mathit{Ground}(\mathcal{P})$ of a program $\mathcal{P}$ is the set of all possible instances of the rules of $\mathcal{P}$.

Example 2. Consider the following program $\mathcal{P}_1$:

$\mathtt{q(1)\,v\,p(2,2).} \qquad \mathtt{q(2)\,v\,p(2,1).}$
$\mathtt{t(X) \text{ :- } q(X), \#sum\{Y : p(X,Y)\} > 1.}$

The instantiation $\mathit{Ground}(\mathcal{P}_1)$ is the following:

$\mathtt{q(1)\,v\,p(2,2).} \qquad \mathtt{q(2)\,v\,p(2,1).}$
$\mathtt{t(1) \text{ :- } q(1), \#sum\{\langle 1 : p(1,1)\rangle, \langle 2 : p(1,2)\rangle\} > 1.}$
$\mathtt{t(2) \text{ :- } q(2), \#sum\{\langle 1 : p(2,1)\rangle, \langle 2 : p(2,2)\rangle\} > 1.}$ □

[5] #`sum` and #`times` applied over an empty set return 0 and 1, respectively.

An *interpretation* for a Datalog$^{\vee\mathcal{A}}$ program $\mathcal{P}$ is a set of standard ground atoms $I \subseteq B_{\mathcal{P}}$. The truth valuation $I(A)$, where A is a standard ground literal or a standard ground conjunction, is defined in the usual way. Besides assigning truth values to the standard ground literals, an interpretation provides the meaning also to (ground) sets, aggregate functions and aggregate literals; the meaning of a set, an aggregate function, and an aggregate atom under an interpretation, is a multiset, a value, and a truth-value, respectively. Let $f(S)$ be a an aggregate function. The valuation $I(S)$ of set S w.r.t. I is the multiset of the first constant of the first components of the elements in S whose conjunction is true w.r.t. I. More precisely,

$$I(S) = [\ t_1 \mid \langle t_1, ..., t_n : Conj \rangle \in S \wedge \text{ Conj is true w.r.t. } I\ ]$$

The valuation $I(f(S))$ of an aggregate function $f(S)$ w.r.t. I is the result of the application of the function f on $I(S)$. (If the multiset $I(S)$ is not in the domain of f, $I(f(S)) = \bot$.)

An aggregate atom $A = Lg \prec_1 f(S) \prec_2 Rg$ is *true w.r.t.* I if: (i) $I(f(S)) \neq \bot$, and, (ii) the relationships $Lg \prec_1 I(f(S))$, and $I(f(S)) \prec_2 Ug$ hold whenever they are present; otherwise, A is false.

Using the above notion of truth valuation for aggregate atoms, the truth valuations of aggregate literals and rules, as well as the notion of model, minimal model, and answer set for Datalog$^{\vee\mathcal{A}}$ are an trivial extension of the corresponding notions in Datalog$^{\vee}$ [12].

2.3 Querying Datalog$^{\vee\mathcal{A}}$ Programs

Let $\mathcal{P}$ be a Datalog$^{\vee\mathcal{A}}$ program and let $\mathcal{F}$ be a set of facts. Then, we denote by $\mathcal{P}_{\mathcal{F}}$ the program $\mathcal{P}_{\mathcal{F}} = \mathcal{P} \cup \mathcal{F}$. Given a query $\mathcal{Q}$ and an interpretation M of $\mathcal{P}$, $\vartheta(\mathcal{Q}, M)$ denotes the set containing each substitution ϕ for the variables in $\mathcal{Q}$ such that $\phi(\mathcal{Q})$ is true in M. The answer to a query $\mathcal{Q}$ over $\mathcal{P}_{\mathcal{F}}$, under the *brave* semantics, denoted by $Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$, is the set $\cup_M \vartheta(\mathcal{Q}, M)$, such that $M \in \mathcal{MM}(\mathcal{P} \cup \mathcal{F})$. The answer to a query $\mathcal{Q}$ over the facts in $\mathcal{F}$, under the *cautious* semantics, denoted by $Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$, is the set $\cap_M \vartheta(\mathcal{Q}, M)$, such that $M \in \mathcal{MM}(\mathcal{P} \cup \mathcal{F}) \neq \emptyset$. If $\mathcal{MM}(\mathcal{P} \cup \mathcal{F}) = \emptyset$, then all substitutions over the universe for variables in $\mathcal{Q}$ are in the cautious answer. Finally, we say that programs $\mathcal{P}$ and P' are *bravely* (resp. *cautiously*) *equivalent* w.r.t. $\mathcal{Q}$, denoted by $\mathcal{P} \equiv_{\mathcal{Q},b} \mathcal{P}'$ (resp. $\mathcal{P} \equiv_{\mathcal{Q},c} \mathcal{P}'$), if for any set $\mathcal{F}$ of facts $Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}}) = Ans_b(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$ (resp. $Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}}) = Ans_c(\mathcal{Q}, \mathcal{P}_{\mathcal{F}})$).

3 Magic-Set Method for Datalog$^{\vee\mathcal{A}}$ Programs

In this section we present the Magic-Set algorithm for Datalog$^{\vee\mathcal{A}}$ programs (short. DMS$^{\mathcal{A}}$), which has been implemented and integrated into the DLV system [19]. Basically, we adopt a strategy for simulating the top-down evaluation

of a query by modifying the original program by means of additional rules, which narrow the computation to what is relevant for answering the query.

The input to the DMS$^{\mathcal{A}}$ algorithm (see Figure 1) is a disjunctive datalog program with aggregate functions $\mathcal{P}$ and a query $\mathcal{Q}$. If the query contains some non-free IDB predicates, it outputs a (optimized) program $\mathtt{DMS}^{\mathcal{A}}(\mathcal{Q},\mathcal{P})$ consisting of a set of *modified* and *magic* rules, stored by means of the sets *modifiedRules*$(\mathcal{Q},\mathcal{P})$ and *magicRules*$(\mathcal{Q},\mathcal{P})$, respectively. The main steps of the algorithm DMS$^{\mathcal{A}}$ are illustrated by means of the following running example, which is an adaptation of the “Strategic Companies” example in [6].

Example 3. We are given a collection C of companies producing some goods in a set G, such that each company $c_i \in C$ is controlled by a set of other companies $O_i \subseteq C$. A subset of the companies $C' \subset C$ is a *strategic set* set if it is a minimal set of companies producing all the goods in G, such that if $O_i \subseteq C'$ for some $i = 1, \ldots, m$ then $c_i \in C'$ must hold. This scenario can be modelled by means of the following program $\mathcal{P}_{sc}$.

$$r_1 : \ \mathtt{sc(C_1)} \ \mathtt{v} \ \mathtt{sc(C_2)} \mathrel{:-} \mathtt{produced_by(P, C_1, C_2)}.$$
$$r_2 : \ \mathtt{sc(C)} \mathrel{:-} \mathtt{controlled_by(C, C_1, C_2, C_3)}, \ \mathtt{sc(C_1)}, \ \mathtt{sc(C_2)}, \ \mathtt{sc(C_3)}.$$

Moreover, a company is dominant if it is strategic and produces only products which are not produced by any other strategic company:

$$r_3 : \ \mathtt{dominant(C)} \mathrel{:-} \mathtt{sc(C)}, \ \#\mathtt{sum}\{\mathtt{P} : \mathtt{produced_by(P, C, C_2)}, \ \mathtt{sc(C_2)}\} = \mathtt{0}.$$

Finally, given a company $\mathtt{c} \in C$, we consider a query $\mathcal{Q}_{sc} = \mathtt{dominant(c)}$. □

The key idea of the algorithm is to materialize binding information which would be propagated during a top-down computation by suitable *adornments.* These are strings of the letters b and f, denoting bound or free for each argument of a predicate. First, adornments are created for query predicates. To efficiently manage adornments, we exploit a stack S of predicates for storing all the adorned predicates to be used for propagating the binding of the query: At each step, an element is removed from S, and each defining rule is processed at a time.

The computation starts in step *2* by initializing the variable *modifiedRules*$(\mathcal{Q},\mathcal{P})$ to the empty set — the need of this structure will be clear in a while. Then, the function ***BuildQuerySeeds*** pushes on the stack S the adorned predicates of $\mathcal{Q}$, and stores in *magicRules*$(\mathcal{Q},\mathcal{P})$ some facts, called *magic seeds.* Each fact in such a variable is the *magic version* of an adorned atom $\mathtt{p}^{\alpha}$ pushed in S, denoted by ***magic***$(\mathtt{p}^{\alpha})$, obtained by eliminating all arguments labelled f in α.

Example 4. Given the query $\mathcal{Q}_{sc} = \mathtt{dominant(c)}$ and the program $\mathcal{P}_{sc}$, ***BuildQuerySeeds*** creates $\mathtt{magic_dominant^{b}(c)}$. and pushes $\mathtt{dominant^{b}}$ onto the stack S. □

Input: A Datalog$^{\vee}$ program $\mathcal{P}$, and a query $\mathcal{Q} = \mathtt{g_1(t_1)}, \ldots, \mathtt{g_n(t_n)}$.
Output: The optimized program $\mathtt{DMS}^{\mathcal{A}}(\mathcal{Q}, \mathcal{P})$.
var S: **stack** of adorned predicates; *modifiedRules*$(\mathcal{Q}, \mathcal{P})$,*magicRules*$(\mathcal{Q}, \mathcal{P})$: **set** of rules;
begin
1. **if** $\mathtt{g_1(t_1)}, \ldots, \mathtt{g_n(t_n)}$ has some IDB predicate **then**
2. *modifiedRules*$(\mathcal{Q}, \mathcal{P})$:=$\emptyset$; $\langle S$, *magicRules*$(\mathcal{Q}, \mathcal{P})\rangle$:=***BuildQuerySeeds***$(\mathcal{Q})$;
3. **while** $S \neq \emptyset$ **do**
4. p^{α}:=S.**pop()**;
5. **for each** rule $r \in \mathcal{P}$: $\mathtt{p(t)} \vee \mathtt{p_1(t_1)} \vee \ldots \vee \mathtt{p_n(t_n)}$:- $\mathtt{q_1(s_1)}, \ldots, \mathtt{q_m(s_m)}$ **do**
6. r_a:=***Adorn***(r_s, p^{α}, S);
7. *magicRules*$(\mathcal{Q}, \mathcal{P})$:= *magicRules*$(\mathcal{Q}, \mathcal{P})$ $\bigcup$ ***Generate***(r_a);
8. *modifiedRules*$(\mathcal{Q}, \mathcal{P})$:= *modifiedRules*$(\mathcal{Q}, \mathcal{P})$ $\bigcup$ $\{$***Modify***$(r_a)\}$;
9. **end for**
10. **end while**
11. $\mathtt{DMS}^{\mathcal{A}}(\mathcal{Q}, \mathcal{P})$:=*magicRules*$(\mathcal{Q}, \mathcal{P})$ $\cup$ *modifiedRules*$(\mathcal{Q}, \mathcal{P})$;
12. **return** $\mathtt{DMS}^{\mathcal{A}}(\mathcal{Q}, \mathcal{P})$;
13. **end if**
end.

Fig. 1. Magic-Set Method for Datalog$^{\vee\mathcal{A}}$ Programs.

3.1 Adornment

The query adornments are then used to propagate their information into the body of the rules defining it, simulating a top-down evaluation. And, in fact, the core of the technique (steps *4*-*9*) consists of removing an adorned predicate $\mathtt{p}^{\alpha}$ from the stack S in step *4*, and in propagating its binding in each (disjunctive) rule r in $\mathcal{P}$ of the form

$$r: \ \mathtt{p(t)} \ \mathtt{v} \ \mathtt{p_1(t_1)} \ \mathtt{v} \ \ldots \ \mathtt{v} \ \mathtt{p_n(t_n)} \ \text{:-} \ \mathtt{q_1(s_1)}, \ \ldots, \ \mathtt{q_m(s_m)}.$$

with $\mathtt{n} \geq 0$, having an atom $\mathtt{p(t)}$ in the head (step *5*).

Obviously various strategies can be pursued concerning the order of processing the body atoms and the propagation of bindings. These are referred to as Sideways Information Passing Strategies (*SIPS*), cf. [4]. Any SIPS must guarantee an iterative processing of all body atoms in r, and simulates the data flow occurring in the top-down evaluation of the query, by iteratively processing all the predicates in r.

Roughly speaking, a SIPS act as follows. Let $\mathtt{q}$ be an atom that has not yet been processed, then its adorned version is created by assuming constants and variables occurring in already considered atoms to be bound, which is denoted by $\mathtt{v} \rightarrow_{\mathtt{X}} \mathtt{q}$, where $\mathtt{X}$ is the set of the variables assumed to be bound which propagate their values into $\mathtt{q}$, and $\mathtt{v}$ is the set of the predicates in which these variables occur. The formal definition of SIPS is provided below.

Definition 1. Let r be a rule having $\mathtt{p}$ in the head, and let $\mathtt{p}^{\alpha}$ be an adornment. A *SIPS* for r is a labelled bipartite graph $\langle V_1 \cup V_2, E\rangle$, where V_1 is the set of subset of $B(r) \cup \{\mathtt{p}^{\alpha}\}$, $V_2 \in B(r)$, and E is a set of arcs satisfying the following conditions:

1. each arc is of the form $v \rightarrow_X s$, where $v \in V_1$ and $s \in V_2$, where X is a non-empty set of variables such that (i) each variable in X appears in s and in either a bound argument position of $\mathtt{p}^{\alpha}$ or a positive body literal of v, and

(ii) for each literal in v there exists a sequence of literals $v = l_0, l_1, ..., l_m = s$ with l_i and l_{i+1} sharing at least a common argument.

2. there exists a total order of $B(r) \cup \{\mathtt{p}^\alpha\}$ in which
   (a) $\mathtt{p}^\alpha$ precedes all members of $B(r)$,
   (b) any literal which does not appear in the graph follows every literal that appears in the graph, and
   (c) for each arc $v \rightarrow_X s$, if $u \in v$ the u precedes s. □

It is well know that if we are able to construct a SIPS for a given rule r and a predicate $\mathtt{p}^\alpha$, then we can use its edges for simulating the data flow from the head to the body of a rule, and, hence, for deriving the adornment of the rule, which is, in fact, performed in the step *6.*

Example 5. Consider the rule $\mathtt{path(X,Y)} \mathbin{:-} \mathtt{path(X,Z)},\ \mathtt{path(Z,Y)}$. together with query $\mathtt{path(1,5)}$?. Then, the adornment of the query predicate, i.e., $\mathtt{path^{bb}(1,5)}$, passes its binding information to $\mathtt{path(X,Z)}$ through $\mathtt{path^{bb}(X,Y)} \rightarrow_{\{\mathtt{X}\}} \mathtt{path(X,Z)}$, which causes the generation of the adorned predicate $\mathtt{path^{bf}(X,Z)}$. Then, we apply $\{\mathtt{path^{bb}(X,Y)},\ \mathtt{path^{bf}(X,Z)}\} \rightarrow_{\{\mathtt{X,Y,Z}\}} \mathtt{path(Z,Y)}$, generating $\mathtt{path^{bb}(Z,Y)}$. The resulting adorned rule is $\mathtt{path^{bb}(X,Y)} \mathbin{:-} \mathtt{path^{bf}(X,Z)},\ \mathtt{path^{bb}(Z,Y)}$. □

We point out that, for each rule, it is possible to derive different SIPS, associated to all the possible permutations of the atoms appearing in the body. The choosing of a strategy does not matter in the case of positive programs, but it represents a serious issue in the case of Datalog$^{\vee\mathcal{A}}$ programs, as shown in the following section.

3.2 Binding Propagation in Datalog$^{\vee\mathcal{A}}$ Programs

Aggregate Atoms. Let us first consider the binding propagation in the presence of aggregate atoms. We recall that an aggregate atom has the form $L_g \leq f\{Vars : Conj\} \leq U_g$, where $Vars$ are variables local w.r.t. the function f, while $Conj$ is a conjunction of literals. All the variables occurring in predicates of $Conjs$ that are not in $Vars$ are said *global* variables.

Since $Conjs$ might contain some variables that are used into other predicates of the rule, we can exploit these variables for propagating the binding into the aggregate atom, too. Then, in the adornment step, literals in $Conj$ can be treated as they were part of the rule; nonetheless some further attention is needed for ensuring the correctness of the SIPS implemented. In fact, literals in $Conj$ have not to be used for propagating bindings to other literals, and, hence, they should be considered at the end of the adornment process. To this aim we extend any standard SIPS, by introducing the additional constraint of preferring for binding propagation aggregate atoms only if there are no other atoms to be processed. Moreover, when only aggregate atoms remain to be processed we prefer the ones having the maximum number of bound variables.

Example 6. Consider again Example 3. When $\texttt{dominant}^{\texttt{b}}$ is removed from the stack, we select rule r_3 for its adornment. Then, C is the unique bound variable and might propagate its binding to both sc(C) or to $\texttt{sc}(\texttt{C}_2)$ trough the fact $\texttt{produced_by}(\texttt{P},\texttt{C},\texttt{C}_2)$. However, non-aggregate atoms are always processed first, and hence $\texttt{dominant}^{\texttt{b}}(\texttt{c})$, passes its binding information to sc(C) through $\texttt{dominant}^{\texttt{b}}(\texttt{C}) \rightarrow_{\{\texttt{C}\}} \texttt{sc}(\texttt{C})$. Then, we apply $\{\texttt{sc}^{\texttt{b}}(\texttt{C}), \texttt{produced_by}(\texttt{P},\texttt{C},\texttt{C}_2)\} \rightarrow_{\{\texttt{C},\texttt{C}_2\}} \texttt{sc}(\texttt{C}_2)$, generating $\texttt{sc}^{\texttt{b}}(\texttt{C}_2)$. The resulting adorned rule is

$$r_{3_a}: \ \texttt{dominant}^{\texttt{b}}(\texttt{C}) \texttt{:-} \texttt{sc}^{\texttt{b}}(\texttt{C}), \ \#\texttt{sum}\{\texttt{P} : \texttt{produced_ by}(\texttt{P},\texttt{C},\texttt{C}_2), \ \texttt{sc}^{\texttt{b}}(\texttt{C}_2)\} = \texttt{0}.$$

and the adorned predicate $\texttt{sc}^{\texttt{b}}$ is pushed on the stack S. □

Disjunctive Programs. Let us now consider the case of disjunctive programs without aggregate functions. Then, as first observed in [14], while in nondisjunctive programs bindings are propagated only head-to-body, any sound rewriting for disjunctive programs has to propagate bindings also head-to-head in order to preserve soundness. Roughly, suppose that a predicate p is relevant for the query, and a disjunctive rule r contains p(X) in the head. Then, besides propagating the binding from p(X) to the body of r (as in the nondisjunctive case), a sound rewriting has to propagate the binding also from p(X) to the other head atoms of r. Consider, for instance, a $\text{Datalog}^{\vee}$ program $\mathcal{P}$ containing rule p(X) v q(Y) :- a(X, Y), r(X). and the query p(1)?. Even though the query propagates the binding for the predicate p, in order to correctly answer the query, we also need to evaluate the truth value of q(Y), which indirectly receives the binding through the body predicate a(X, Y). For instance, suppose that the program contains facts a(1, 2), and r(1); then atom q(2) is relevant for query p(1)? (i.e., it should belong to the magic set of the query), since the truth of q(2) would invalidate the derivation of p(1) from the above rule, because of the minimality of the semantics.

It follows that, while propagating the binding, the head atoms of disjunctive rules must be all adorned as well. We achieve this by defining an extension of any non-disjunctive SIPS to the disjunctive case. The constraint for such a disjunctive SIPS is that head atoms (different from p(t)) cannot provide variable bindings, they can only *receive* bindings (similarly to negative literals in standard SIPS). So they should be processed only once all their variables are bound or do not occur in yet unprocessed body atoms.[6] Moreover they cannot make any of their free-variables bound.

The function ***Adorn*** produces an adorned disjunctive rule from an adorned predicate and a suitable unadorned rule by employing the refined SIPS, pushing all newly adorned predicates onto S. Hence, in step *6* the rule r_a is of the form

$$r_a: \ \texttt{p}^{\alpha}(\texttt{t}) \ \texttt{v} \ \texttt{p}_1^{\alpha_1}(\texttt{t}_1) \ \ldots \ \texttt{p}_n^{\alpha_n}(\texttt{t}_n) \texttt{:-} \texttt{q}_1^{\beta_1}(\texttt{s}_1), \ldots, \texttt{q}_m^{\beta_m}(\texttt{s}_m).$$

[6] Recall that the safety constraint guarantees that each variable of a head atom also appears in some positive body-atom.

Example 7. Consider again Example 3. When $\mathtt{sc}^{\mathtt{b}}$ is removed from the stack, we first select rule r_1 and the head predicate $\mathtt{sc}(\mathtt{C}_1)$. Then, the adorned version is

$$r'_{1_a}:\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1)\ \mathtt{v}\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2)\ \mathtt{:\!-}\ \mathtt{produced_by}(\mathtt{P},\mathtt{C}_1,\mathtt{C}_2).$$

Next r_1 is processed again, this time with head predicate $\mathtt{sc}(\mathtt{C}_2)$, producing

$$r''_{1_a}:\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2)\ \mathtt{v}\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1)\ \mathtt{:\!-}\ \mathtt{produced_by}(\mathtt{P},\mathtt{C}_1,\mathtt{C}_2).$$

and finally, processing r_2 we obtain

$$r_{2_a}:\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C})\ \mathtt{:\!-}\ \mathtt{controlled_by}(\mathtt{C},\mathtt{C}_1,\mathtt{C}_2,\mathtt{C}_3),\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1),\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2),\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_3).$$ □

3.3 Generation

The algorithm uses the adorned rule r_a for generating and collecting the *magic rules* in step *7*, which simulate the top-down evaluation scheme. Since r_a is in general a disjunctive rule with aggregate atoms, ***Generate*** first produces a non-disjunctive intermediate rule, say r'_a by moving head atoms into the body and by replacing each aggregate atom, say $L_g \leq f\{Vars : Conj\} \leq U_g$, by the conjunction $Conj$.

Then, for each adorned atom $\mathtt{p}$ in the body of an adorned rule r'_a, a magic rule r_m is generated such that (i) the head of r_m consists of ***magic***($\mathtt{p}$), and (ii) the body of r_m consists of the magic version of the head atom of r'_a, followed by all of the predicates of r'_a which can propagate the binding on $\mathtt{p}$.

Example 8. In the program of Example 6, from the rule r'_3, we first derive the following standard rule

$$\mathtt{dominant}^{\mathtt{b}}(\mathtt{C})\ \mathtt{:\!-}\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}),\ \mathtt{produced_by}(\mathtt{P},\mathtt{C},\mathtt{C}_2),\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2).$$

and, then, the magic rules

$$\mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C})\ \mathtt{:\!-}\ \mathtt{magic_dominant}^{\mathtt{b}}(\mathtt{C}),\ \mathtt{produced_by}(\mathtt{P},\mathtt{C},\mathtt{C}_2),\ \mathtt{st}^{\mathtt{b}}(\mathtt{C}_2).$$
$$\mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C}_2)\ \mathtt{:\!-}\ \mathtt{magic_dominant}^{\mathtt{b}}(\mathtt{C}),\ \mathtt{produced_by}(\mathtt{P},\mathtt{C},\mathtt{C}_2),\ \mathtt{st}^{\mathtt{b}}(\mathtt{C}_2).$$

Similarly, by looking at Example 7, from the rule r'_{1_a} first its non-disjunctive intermediate rule

$$\mathtt{sc}^{\mathtt{b}}(\mathtt{C}_1)\ \mathtt{:\!-}\ \mathtt{sc}^{\mathtt{b}}(\mathtt{C}_2),\mathtt{produced_by}(\mathtt{P},\mathtt{C}_1,\mathtt{C}_2).$$

is produced, from which the magic rule

$$\mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C}_2)\ \mathtt{:\!-}\ \mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C}_1),\ \mathtt{produced_by}(\mathtt{P},\mathtt{C}_1,\mathtt{C}_2).$$

is generated. Similarly, from the rule r''_{1_a} we obtain

$$\mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C}_1)\ \mathtt{:\!-}\ \mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C}_2),\ \mathtt{produced_by}(\mathtt{P},\mathtt{C}_1,\mathtt{C}_2).$$

and finally r_{2_a} gives rise to the following rules

$$\mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C}_1)\ \mathtt{:\!-}\ \mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C}),\ \mathtt{controlled_by}(\mathtt{C},\mathtt{C}_1,\mathtt{C}_2,\mathtt{C}_3).$$
$$\mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C}_2)\ \mathtt{:\!-}\ \mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C}),\ \mathtt{controlled_by}(\mathtt{C},\mathtt{C}_1,\mathtt{C}_2,\mathtt{C}_3).$$
$$\mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C}_3)\ \mathtt{:\!-}\ \mathtt{magic_sc}^{\mathtt{b}}(\mathtt{C}),\ \mathtt{controlled_by}(\mathtt{C},\mathtt{C}_1,\mathtt{C}_2,\mathtt{C}_3).$$

□

3.4 Modifications

In step *8* the *modified rules* are generated and collected. These rules represent the rewriting of the original program in which the instantiation of body predicates is limited by the magic predicates. Specifically, the function ***Modify*** constructs a rule of the following form

$$\mathtt{p(t)} \vee \mathtt{p_1(t_1)} \vee \ldots \vee \mathtt{p_n(t_n)} \text{ :- } \boldsymbol{magic}(\mathtt{p}^{\alpha}(\mathtt{t})),\ \boldsymbol{magic}(\mathtt{p}_1^{\alpha_1}(\mathtt{t_1})), \ldots,\ \boldsymbol{magic}(\mathtt{p}_n^{\alpha_n}(\mathtt{t_n})),\ \mathtt{q_1(s_1)}, \ldots, \mathtt{q_m(s_m)}.$$

Finally, after all the adorned predicates have been processed the algorithm outputs the program $\mathtt{DMS}^{\mathcal{A}}(\mathcal{Q}, \mathcal{P})$.

Example 9. In our running example, we derive the following set of modified rules:

$$\begin{aligned}
r'_{1_m} &: \mathtt{sc(C_1)\ v\ sc(C_2)} \text{ :- } \mathtt{magic_sc^b(C_1),\ magic_sc^b(C_2),\ produced_by(P, C_1, C_2).} \\
r''_{1_m} &: \mathtt{sc(C_2)\ v\ sc(C_1)} \text{ :- } \mathtt{magic_sc^b(C_2),\ magic_sc^b(C_1),\ produced_by(P, C_1, C_2).} \\
r_{2_m} &: \mathtt{sc(C)} \text{ :- } \mathtt{magic_sc^b(C),\ controlled_by(C, C_1, C_2, C_3),\ sc(C_1),\ sc(C_2),\ sc(C_3).} \\
r_{3_m} &: \mathtt{dominant(C)} \text{ :- } \mathtt{magic_dominant^b(C),\ sc^b(C),} \\
&\qquad \#\mathtt{sum}\{\mathtt{P : produced_by(P, C, C_2),\ sc^b(C_2)}\} = 0.
\end{aligned}$$

where r'_{1_m} (resp. r''_{1_m}, r_{2_m}, r_{3_m}) is derived by adding magic predicates and stripping off adornments for the rule r'_{1_a} (resp. r''_{1_a}, r_{2_a}, r_{3_a}). Thus, the optimized program $\mathtt{DMS}^{\mathcal{A}}(\mathcal{Q}_{sc}, \mathcal{P}_{cs})$ comprises the above modified rules as well as the magic rules in Example 8, and the magic seed $\mathtt{magic_dominant^b(c)}$. □

We conclude the exposition of this algorithm by stressing that the rewriting computed throughout its application is, in fact, an equivalent rewriting of the input program, in the sense provided by the following proposition.

Theorem 1 (Soundness of the DMS$^{\mathcal{A}}$ Algorithm). *Let $\mathcal{P}$ be a Datalog$^{\vee}$ program, let $\mathcal{Q}$ be a query. Then,* $\mathtt{DMS}^{\mathcal{A}}(\langle \mathcal{Q}, \mathcal{P} \rangle) \equiv_{\mathcal{Q},b} \mathcal{P}$ *and* $\mathtt{DMS}^{\mathcal{A}}(\langle \mathcal{Q}, \mathcal{P} \rangle) \equiv_{\mathcal{Q},c} \mathcal{P}$ *hold.*

4 Conclusions

Motivated by the application in data integration settings, we have presented a technique for the optimization of (partially) bound queries that extends the Magic-Set method to the case of disjunctive programs with aggregate operators. The technique has been fully implemented into the DLV system.

We point out that our investigation can be of a great interest in several other applicative domains. In fact, aggregate functions in logic programming languages appeared already in the 80s, when their need emerged in deductive databases like LDL. Currently, they are supported in the Smodels system, besides DLV, and their importance in knowledge representation tasks is widely recognized, since they can be simulated only by means of inefficient and unnatural encodings of

the problems. As an example, suppose that a user wants to know if the sum of the salaries of the employees working in a team exceeds a given budget. To this end, the user should first order the employees defining a successor relation. Then she should define a *sum* predicate, in a recursive way, which computes the sum of all salaries, and compare its result with the given budget. This approach has two drawbacks: (1) It is bad from the KR perspective, as the encoding is not natural at all; (2) It is inefficient, as the (instantiation of the) program is quadratic (in the cardinality of the input set of employees).

Concerning future work, our objective is to extend the Magic-Set method to the case of disjunctive programs with constraints and unstratified negation, such that it can be fruitfully applied on arbitrary DLV programs. We believe that the framework developed in this paper is general enough to be extended to these more involved cases.

Acknowledgments

The research was supported by the European Commission under the INFOMIX project (IST-2001-33570).

References

1. O. Arieli, M. Denecker, B. Van Nuffelen, and M. Bruynooghe. Database repair by signed formulae. In *Proc. of FoIKS'04*, volume 2942 of *LNCS*, pp. 14–30. Springer, February 2004.
2. F. Bancilhon, D. Maier, Y. Sagiv, and J.D. Ullman. Magic Sets and Other Strange Ways to Implement Logic Programs. In *Proc. of PODS'86*, pp. 1–16, 1986.
3. P. Baumgartner, U. Furbach, and I. Niemelä. Hyper Tableaux. In *Proc. of JELIA'96*, number 1126 in LNCS, pp. 1–17. Springer, 1996.
4. C. Beeri and R. Ramakrishnan. On the power of magic. *JLP*, 10(1–4):255–259, 1991.
5. A. Behrend. Soft stratification for magic set based query evaluation in deductive databases. In *Proc. of PODS'03*, pp. 102–110. ACM Press, 2003.
6. M. Cadoli, T. Eiter, and G. Gottlob. Default Logic as a Query Language. *TKDE*, 9(3):448–463, 1997.
7. A. Calì, D. Lembo, and R. Rosati. Query rewriting and answering under constraints in data integration systems. In *Proc. 18th Int'l Joint Conference on Artificial Intelligence (IJCAI 2003)*, pp. 16–21, 2003.
8. C. Cumbo, W. Faber, and G. Greco. Improving Query Optimization for Disjunctive Datalog. In *Proc. of the Joint Conference on Declarative Programming APPIA-GULP-PRODE 2003*, pp. 252–262, 2003.
9. T. Dell'Armi, W. Faber, G. Ielpa, N. Leone, and G. Pfeifer. Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In *Proc. 18th Int'l Joint Conference on Artificial Intelligence (IJCAI 2003)*, pp. 847–852, 2003.
10. T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. *Logic-Based Artificial Intelligence*, pp. 79–103. Kluwer, 2000.

11. T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *TODS*, 22(3):364–418, September 1997.
12. M. Gelfond and V. Lifschitz. Classical Negation in Logic Programs and Disjunctive Databases. *New Generation Computing*, 9:365–385, 1991.
13. G. Greco, S. Greco, and E. Zumpano. A logic programming approach to the integration, repairing and querying of inconsistent databases. In P. Codognet, editor, *Proc. 17th Int'l Conference on Logic Programming (ICLP 2001)*, LNCS 2237, pp. 348–364. Springer, 2001.
14. S. Greco. Binding Propagation Techniques for the Optimization of Bound Disjunctive Queries. *IEEE TKDE*, 15(2):368–385, March/April 2003.
15. S. Greco, D. Saccà, and C. Zaniolo. The PushDown Method to Optimize Chain Logic Programs (Extended Abstract). In *ICALP'95*, pp. 523–534, 1995.
16. A. Gupta and I.S. Mumick. Magic-sets Transformation in Nonrecursive Systems. In *Proc. of PODS'92*, pp. 354–367, 1992.
17. T. Janhunen, I. Niemelä, P. Simons, and J.-H. You. Partiality and Disjunctions in Stable Model Semantics. In *Proc. of KR'00*, April 12-15, Breckenridge, Colorado, USA, pp. 411–419. Morgan Kaufmann.
18. D.B. Kemp, D. Srivastava, and P.J. Stuckey. Bottom-up evaluation and query optimization of well-founded models. *Theoretical Computer Science*, 146:145–184, July 1995.
19. N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The DLV System for Knowledge Representation and Reasoning. *to appear* in ACM TOCL.
20. I.S. Mumick, S.J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic is relevant. In *Proc. of SIGMOD'00*, pp. 247–258, 1990.
21. R. Ramakrishnan, Y. Sagiv, J.D. Ullman, and M.Y. Vardi. Logical Query Optimization by Proof-Tree Transformation. *JCSS*, 47(1):222–248, 1993.
22. P. Seshadri, J.M. Hellerstein, H. Pirahesh, T.Y.C. Leung, R. Ramakrishnan, D. Srivastava, P.J. Stuckey, and S. Sudarshan. Cost-based optimization for magic: Algebra and implementation. In *Proc. of SIGMOD'96*, pp. 435–446. ACM Press, June 1996.
23. P.J. Stuckey and S. Sudarshan. Compiling query constraints. In *Proc. of PODS'94*, pp. 56–67. ACM Press, May 1994.
24. J. D. Ullman. *Principles of Database and Knowledge Base Systems.*

Checking the Completeness of Ontologies: A Case Study from the Semantic Web*

Valentina Cordì and Viviana Mascardi

Dipartimento di Informatica e Scienze dell'Informazione – DISI,
Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy.
1996s081@educ.disi.unige.it, mascardi@disi.unige.it

Abstract. The paper discusses a formal framework for proving correctness and completeness of ontologies during its life-cycle. We have adopted our framework for the development of a case study drawn from the Semantic Web. In particular we have developed an ontology for content-based retrieval of XML documents in Peer-to-peer networks.

1 Introduction

Peer-to-peer (P2P) systems [10] have emerged as a promising new paradigm for distributed computing, as witnessed by the experience with Napster and Gnutella and by the growing number of research events related to them. Current P2P systems focus strictly on handling semantic-free, large-granularity requests for objects by identifier (typical name), which both limits their usability and restricts the techniques that might be employed to access data. Intelligent agents that exploit ontologies to perform content-based information retrieval in P2P networks may represent a viable solution to overcome the limitations of current P2P networks [11,1].

A recent proposal for a semantic, policy-based system for the retrieval of XML documents in P2P networks comes from [9], where peers are organised into thematic groups coordinated by a "super-peer agent" that exploits a "group ontology" to set the concepts managed by the group. The focus of [9] is on the architecture of the system; the engineering stages that a developer must follow in order to design, build and evaluate the group ontology are not addressed at all.

Developing an ontology is akin to defining a set of data and their structure for other programs to use. Problem-solving methods, domain-independent applications, and software agents use ontologies and knowledge bases built from ontologies as data. The engineering stages that an ontology undergoes during its life-cycle include its evaluation with respect to general and domain-dependent requirements. In particular, proving the ontology completeness and consistency is a very important step to face in order to develop correct, re-usable and maintainable ontologies. From a logical point of view, completeness is a property associated with combining a procedure for constructing well-formed formulas, a definition of truth that relates to interpretations and models of logical systems, and a proof procedure that allows new well-formed formulas to be

* Parts of this document appear in [3].

derived from old ones. A logical system is logically complete if every true well-formed formula can be derived. The other side to logical completeness is consistency. If falsity can be derived, then any well-formed formula can be derived, so trivially all true well-formed formulas can be derived.

When talking about ontologies, completeness and consistency assume a different meaning, although the conceptual relation with their logical counterparts is usually respected. While the meaning of consistency w.r.t. ontologies is pretty simple – the ontology should not contain conflicting information – there are different definitions of ontological completeness.

According to Colomb and Weber [2], an information system has the *potential* of being "ontologically complete" if it matches the social reality of the organisation in which the system is embedded. The potential for completeness, which is analogous to logical and computational completeness, has been called "ontological adequacy" by Guarino [6]. Colomb and Weber propose a set of guidelines for checking the ontological completeness of information systems. Fox and Grüninger [4] define the "functional completeness" of an ontology as its ability to represent the information necessary for a function to perform its task. They also propose a set of theorems that state under which conditions an ontology is complete [5].

All the authors that deal with the problem of checking the completeness of an ontology w.r.t. its requirements, agree that this check should be designed in such a way to be easily automatised and computationally tractable. In this paper, we provide a notion of completeness based on [5] but simpler than that, and whose check can be partially automatised. Both the notion we propose and the framework for proving the completeness of ontology we have developed are based on computational logic.

We have adopted our framework for the development of a case study drawn from the Semantic Web, where proving the completeness of an ontology can be crucial for safety and security reasons.

The structure of the paper is the following: Section 2 introduces the case study based on [9]. Section 3 introduces some techniques from the literature and then explains our formal framework for proving completeness of ontologies. Section 4 shows the development of the case study emphasising the ontology evaluation by means of our framework. Conclusions follow.

2 The case study: describing and retrieving XML documents

To show how our formal framework for proving the completeness of ontologies works, we consider a scenario simpler than that for which we need to develop the "real" ontology, namely the P2P network described in [9]. There, peers are organised into thematic groups, each one coordinated by a "super-peer agent". The super-peer agent provides an ontology ("group ontology") that sets the concepts dealt with by the group and establishes the relationships among them. Each peer can dynamically enter and leave any group inside the P2P network. When the peer joins a group for the first time, it is requested to provide to the super-peer agent as much information as possible about the concepts that are dealt with by the documents it is willing to share. This allows the super-peer agent to know which peers are more likely to deal with which concepts.

When a query is submitted to a peer, the peer forwards it to the super-peer which can understand the meaning of the terms appearing in the query by exploiting the group ontology. Since the super-peer knows which peers deal with which concepts, it identifies the peers in the group that can contain an answer for the query and forwards the query only to them, in order to minimise the number of messages exchanged inside the group.

The simplified scenario that we consider involves the development of a system able to support the automatic classification of XML documents retrieved from the network (just a binary classification: "is the XML document talking about a given topic or not?"). In particular the case study faces the development of an ontology for structuring the knowledge about XML documents talking about movies. In this simplified scenario, formally demonstrating the completeness of the ontology is not a very critical issue and we perform this demonstration mainly for illustrative purposes. Nevertheless, there are many real situations drawn from the Semantic Web domain where this formal proof *must* be carried out for safety and security reasons.

The knowledge about XML documents talking about movies is based on:

1. the semantics of tags that appear in the XML document, and
2. the XML document structure.

For example, both documents in Table 1 describe a movie, even if they are characterised by different structure and different tags.

```
<movie>
  <title>Title1<title>
  <actors><actor>Act1</actor>
          <actor>Act2</actor>
  </actors>
  <directed_by><name>Name</name>
         <surname>Surn</surname>
  </directed_by>
</movie>
```

```
<film>
  <title>Title2<title>
  <actors>Act3, Act4</actors>
  <director>Dir</director>
</film>
```

Table 1. Two XML documents dealing with movies

As far as tags are concerned, the first document uses <movie> to refer to a movie, while the second document uses <film>. In this context, the semantics of "movie" and "film" is the same. The director is identified by the tag <directed_by> in the first document, and by the tag <director> in the second one. Again, despite to their syntactic difference, these two tags have the same semantics.

As far as the structure is concerned, the tag <actors> is structured into a list of <actor> and the tag <directed_by> is composed by <name> and <surname> in the first document, while the corresponding tags in the second document contain strings.

The prototypical ontology must contain all the information needed to classify XML documents talking about movies. In particular, it must contain the information that:

– the tags `<film>` and `<movie>`, and `<director>` and `<directed_by>` represent the same concepts in the movie context;
– `<actors>` can contain a string or a list of `<actor>` tags;
– the director, be it identified by `<directed_by>` or by `<director>`, may contain a string or a structure including `<name>` and `<surname>`.

In order to build the ontology, we retrieved a set of existing XML documents dealing with movies from the web and we manually analysed each of them in order to identify the structural and the semantic rules exemplified above. Some documents we used for our purposes are http://catcode.com/cit041x/assignment4a.html, http://www-db.stanford.edu/pub/movies/mains218.xml, and http://www.flixml.org/flixml/detour.xml.

The purpose was to build an ontology which could tell that a document starting with the tag `<film>` or `<movie>` (and others, that we do not discuss here), and containing somewhere a tag `<director>` or `<directed_by>`, possibly with different content, is likely to talk about movies. Given a new XML document, the ontology should allow to answer "yes, it talks about movies because it matches the semantic and structural rules" or "no, it does not talk about movies".

3 A formal framework for proving completeness of ontologies

Our formal framework is based on the work on TOVE by Grüninger and Fox [5]. TOVE is a methodology based on experiences in the development of TOVE (Toronto Virtual Enterprise).

The TOVE approach to ontology development starts with the definition of the motivating scenarios that arise in the applications. Such scenarios may be presented by industrial partners as problems which they encounter in their enterprises. The motivating scenarios often have the form of story problems or examples which are not adequately addressed by existing ontologies.

Given the motivating scenarios, a set of queries will arise which place demands on an underlying ontology. These queries can be considered as the requirements that are in the form of questions that an ontology must be able to answer. These are called the *informal competency questions*, since they are not yet expressed in the formal language of the ontology.

Once informal competency questions have been defined, they should be restated using some formal language suitable for expressing the ontology terminology. This activity is carried out manually. The ontology terminology must be able to correctly and easily represent the objects in the domain of discourse as constants and variables in the language. Attributes of objects may be defined by unary predicates; relations among objects may be defined using n-ary predicates. The two languages that Grüninger and Fox suggest for expressing both the ontology terminology and the formal competency questions are first-order logic and KIF [13].

In [4], the concepts of competency and completeness of an ontology are informally stated:

Given a properly instantiated model of an enterprise and an accompanying theorem prover (perhaps Prolog or a deductive database), the competence of an ontology is the set of queries that it can answer. [. . .]
The Functional Completeness of an ontology is determined by its competency, i.e., the set of queries it can answer with a properly instantiated model. Given a particular function (application), its enterprise modelling needs can be specified as a set of queries. If these queries can be "reduced to"[1] the set of competency questions specified for the chosen ontology, then the ontology is sufficient to meet the modelling needs of the application.

This informal statement corresponds to the formal definition provided by the completeness theorems discussed in [5]. These theorems have one of the following forms, where $T_{ontology}$ is the set of axioms in the ontology, T_{ground} is a set of ground literals (instances), Q is a first-order sentence specifying the query in the competency question, and Φ is a set of first-order sentences defining the set of conditions under which the solutions to the problem are complete:

- $T_{ontology} \cup T_{ground} \models \Phi$ if and only if $T_{ontology} \cup T_{ground} \models Q$.
- $T_{ontology} \cup T_{ground} \models \Phi$ if and only if $T_{ontology} \cup T_{ground} \cup Q$ is consistent.
- $T_{ontology} \cup T_{ground} \cup \Phi \models Q$ or $T_{ontology} \cup T_{ground} \cup \Phi \models \neg Q$.
- All models of $T_{ontology} \cup T_{ground}$ agree on the extension of some predicate P.

Completeness theorems can also provide a means of determining the extendibility of an ontology, by making explicit the role that each axiom plays in proving the theorem. Any extension to the ontology must be able to preserve the completeness theorems.

Starting from Grüninger and Fox's definitions, and integrating suggestions coming from other methodologies such as EXPLODE [7], "A Guide to Creating your First Ontology" [12] (in the following identified by *OD101* for readability), and Uschold's "Unified Method" [14] (in the following identified by *UniMeth*), we define our guidelines for developing a complete ontology. UniMeth embraces TOVE and the Enterprise methodology [15] in a unique framework. For this reason, in the following we will refer to UniMeth instead of specifically referring to TOVE.

The engineering stages that an ontology developer should follow according to our integrated approach, fully discussed in [3], are:

- **Domain analysis.** This development stage can be faced by answering the questions that EXPLODE, OD101 and UniMeth suggest, such as which are the expected users of the methodology and which are the ontology domain and extended purpose. UniMeth also suggests to identify fairly general scenarios and use them to help clarify specific uses of the ontology.
- **Requirement definition.** EXPLODE, OD101 and UniMeth all suggest to identify the competency questions. Besides competency questions, UniMeth allows the developer to use other techniques for the extraction of the ontology requirements such as defining the detailed motivating scenarios, brainstorming and trimming.

[1] By reducible, Fox and Grüninger mean that the questions can be re-written using the objects provided by the chosen ontology.

EXPLODE also suggests to clearly identify the specific constraints from the hardware/software system that come from other modules in the system that interact with the ontology.

- **Informal specification of the ontology.** This step can be faced by identifying the most important terms of the ontology and using an ontology-editing environment to graphically represent the ontology concepts and the relations among them.
- **Formal specification of the ontology.** Following UniMeth, in this step we suggest to use definite Horn Clauses as the formal language for defining the ontology. We refer to the set of Horn Clauses specifying the ontology as $Progr_{ontology}$.
- **Testing, validation, verification.** The primary validation technique that all the methodologies support consists of informally checking the ontology against the competency questions. By performing this check, it may be realised for example that some motivating scenarios were not correctly addressed. Previous choices can then be adjusted and corrected.
- **Completeness check.** According to UniMeth and to our suggestion for formally specifying the ontology, the developer should manually restate the informal competency questions as goals (negative Horn clauses) and should demonstrate that for each competency question restated as a goal, Q_{Goal}, there exists a refutation for $Progr_{ontology} \cup Q_{Goal}$ [8]. Obviously, this can be automatised by using any Prolog interpreter or compiler to demonstrate that the goal Q_{Goal} succeeds if called within the Prolog program $Progr_{ontology}$. In this sense, the approach to completeness check that we propose is "partially automatised": once the ontology and the informal competency questions have been manually restated as Prolog programs and goals, the completeness check can be performed in a completely automatic way.
- **Other engineering steps.** Before the goal of developing an ontology can be considered achieved, other engineering steps must be faced besides the demonstration of its completeness. EXPLODE, OD101 and UniMeth suggest to face:
  - the development of intermediate prototypes;
  - the iterative refinement of previous choices, according to the outcomes of the completeness check and of the prototype execution;
  - the implementation of a machine-readable ontology;
  - the meetings with clients to perform an iterative check; and
  - the production of documentation on the ontology and on its development process.

  These steps are not a central issue in this paper, so we will not face them. The reader can refer to [3] for details.

4 Developing a complete ontology for the retrieval of XML documents on movies

In this section we discuss the stages – from the domain analysis to the check of the ontology completeness – that we followed to develop the ontology introduced in Section 2.

– **Domain analysis.**
Since the ontology under development is just a toy-example, the answers to the questions suggested by OD101 and UniMeth for analysing the domain are not very meaningful: there are no expected users of the methodology, the domain is the one described in Section 2, and the intended purpose of the ontology is to provide a test-bed for evaluating our approach to completeness check. UniMeth suggests to identify fairly general scenarios and use them to help clarify specific uses of the ontology. For example, being able to recognise both documents in Table 1 as documents dealing with movies is a general motivating scenario for our ontology.
– **Requirement definition.**
We have identified the following competency questions for our ontology.
1. **Competency Question:** Should the ontology be able to separate the concepts related to the document structure from those related to the document semantics?
Expected answer: Yes, it should. This separation is very important because it will allow re-using the ontology to classify documents in domains different from movies, only requiring an extension to the ontology concepts related with the document semantics.
2. **Competency Question:** What are the syntactic equivalent representations of the tag "title" in the context of tag "heading"?
Expected answer: The representations of "title" in the context of "heading" are 't', 'Title', 'title', 'TITLE', 'movieTitle', 'titleMovie'.
3. **Competency Question:** What is the meaning of the tag "title" in the context of the tag "heading"?
Expected answer: The meaning is the "title" element (which is different from the "title" tag).
4. **Competency Question:** Can the tag "actors" contain either a string or a list of "actor" tags?
Expected answer: yes, it can.
5. **Competency Question:** Can the information about the title of the film be an attribute of the tag "movie"?
Expected answer: yes, it can.

Besides using the competency questions, UniMeth suggests to define the detailed motivating scenarios that include possible solutions to the problem addressed by the ontology. A motivating scenario for our ontology is that it must be able to classify the documents whose fragments are shown in Tables 2 and 3, as well as other documents that we downloaded from `http://www-db.stanford.edu/pub/movies/` and `http://catcode.com/cit041x/assignment4a.html`, as movie documents.
EXPLODE suggests to clearly identify the specific constraints from the hardware/software system that come from other modules in the system that interact with the ontology. Our ontology will be used by intelligent agents that help the peers in a P2P network in deciding which of the XML documents they are willing to share deal with movies, and which do not. The modules that the ontology will interact with are those described in [9]. Assuming that all the documents shared by peers in a P2P network have a common structure is not realistic: peers share documents

```
<title role="main"> Detour </title>
<releaseyear role="initial"> 1945 </releaseyear>
<language> English </language>
<studio> PRC (Producers Releasing Corporation) </studio>
<cast> <leadcast>
     <male id="TN"> T. Neal <role> Al </role> </male>
     <female id="AS"> A. Savage <role> Vera </role> </female>
   </leadcast>
   <othercast> <male> .... </male> ....
   </othercast> </cast>
<crew><director>Edgar G. Ulmer</director> ....
```

Table 2. A fragment of `http://www.flixml.org/flixml/detour.xml`

```
<fid> SMg10 </fid>
<t> Bridget Jones's Diary </t>
<year> 2001 </year>
<dirs> <dir> <dirk> R </dirk><dirn> NancyMeyer </dirn> </dir>
```

Table 3. A fragment of `http://www-db.stanford.edu/pub/movies/mains218.xml`

characterised by very different structures and very different tags. An ontology that tries to conciliate these differences can prove extremely useful in this context. In order to be used in a real P2P application, our ontology should be extended to deal with other subjects besides movies (hence, the requirement that the syntactic and the semantic aspects are clearly separated in the ontology).

- **Informal specification of the ontology.**
  Following OD101, we identified the most important terms of the ontology. For example, the terms “tag”, “attribute”, “movie”, “title”, “year” must be represented. Afterwards, we used an ontology-editing environment to graphically represent the ontology concepts and the relations among them.
  Figure 1 represents the hierarchy of concepts that belong to our ontology. The ontology was edited using Protégé 2.0, a drawing tool developed by the Stanford University.
  Note that semantic aspects are separated from syntactic ones. The former are collected under the general concept “Element”, while the latter are collected under the “Tag” concept. The relationship between tags and elements is that a tag has a context, which may be the root of the document or another tag, and a meaning, which is an element.
  The hierarchy of concepts alone is not enough informative. In order to make the ontology useful and complete with respect to its requirements, we had to describe the internal structure of concepts. For example, Figure 2 shows the attributes of the concept Movie.
- **Formal specification of the ontology.**
  The ontology graphically represented in Figures 1 and 2 can be also represented using definite Horn clauses.

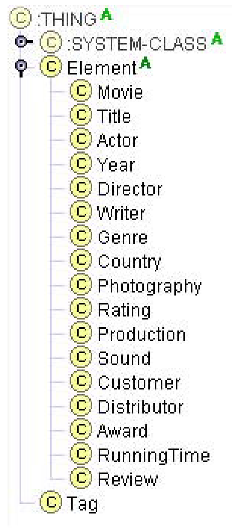


Fig. 1. Concept hierarchy

Movie (type=:STANDARD-CLASS)

Name: Movie

Role: Concrete

Documentation

Constraints

Template Slots

Name	Type	Cardinality	Other Facets
title	Class	required single	parents={Title}
actor	Class	multiple	parents={Actor}
year	Class	required single	parents={Year}
director	Class	required single	parents={Director}
writer	Class	required single	parents={Writer}
genre	Class	single	parents={Genre}
country	Class	single	parents={Country}
photography	Class	single	parents={Photography}
rate	Class	single	parents={Rating}
production	Class	required single	parents={Production}
sound	Class	single	parents={Sound}
customer	Class	single	parents={Customer}
distributor	Class	required single	parents={Distributor}
award	Class	multiple	parents={Award}
runningTime	Class	single	parents={RunningTime}
review	Class	multiple	parents={Review}

Fig. 2. The attributes of the concept “Movie”

We can represent the hierarchy of concepts in a standard way by means of the `isA` relation as shown in Table 4. We adopt a Prolog-like syntax for Horn clauses; text preceded by one or more "`%`" is a comment.

```
%%% THING CONCEPT %%%
isA(tag, thing).
isA(element, thing).

%%% ELEMENT CONCEPT %%%%
isA(movie, element).
isA(title, element).
isA(actor, element).
isA(year, element).
.........
```

Table 4. Formal specification of the ontology: isA relation

The instances of a concept can be defined by means of an `instanceOf` relation which has an instance of a concept and the concept to which the instance belongs as its arguments. Instances are represented by the functor `instance` plus a set of arguments which represents the attributes of the instance. For example, tags are identified by an atom (the tag identifier), another atom (the identifier of the tag context[2]), an element (the tag meaning) and a list of strings (all the possible syntactic representations of the tag inside the XML document). As shown by the first clause of Table 5, an instance of the `title` tag may have an `heading` context (2nd argument, this argument must be an instance of a tag), the `title` meaning (3rd argument; this argument must be an element), and the list of `['t', 'Title', 'title', 'TITLE', 'movieTitle', 'titleMovie']` syntactic representations (4th argument).
Another instance of the `title` tag may have the same arguments as the previous one except for the context, which may be `movie` (second clause of Table 5). This means that two XML documents where the tag `title` appears as a sub-element or an attribute of either the `heading` tag or the `movie` tag can be both considered documents that represent movies. Other examples of instances are included in Table 5.
Definite Horn clauses can be also used to express consistency constraints on the structure of the ontology. For example, the clause shown in Table 6 is an axiom stating that the context of a tag must be a tag and that the semantics of a tag must be an element.

– **Testing, validation, verification.**
The primary validation technique that all the methodologies support consists of checking the ontology against the informal competency questions. One of the com-

[2] The tag A is in the context of the tag B if either A is an attribute of B or if it is a sub-element of B.

```
%%% TAG TITLE %%%
instanceOf(instance(title, heading, title,
['t', 'Title', 'title', 'TITLE',
'movieTitle', 'titleMovie']), tag).

instanceOf(instance(title, movie, title,
['t', 'Title', 'title', 'TITLE',
'movieTitle', 'titleMovie']), tag).

%%% TAG ACTOR %%%
instanceOf(instance(actor, actors, actor,
['actor', 'ACTOR', 'Actor']), tag).

%%% TAG ACTORS %%%
instanceOf(instance(actors, credits, actor,
['actors', 'ACTOR', 'Actor', 'cast', 'Cast', 'CAST']), tag).

.......
```

Table 5. Formal specification of the ontology: `instanceOf` relation

```
instanceOf(instance
              (TagId, TagContext,
               TagSemantics, TagSyntax),
           tag) :-
instanceOf(instance(TagContext, _, _, _), tag),
isA(TagSemantics, element).
```

Table 6. Formal specification of the ontology: consistency axioms

petency questions we identified for the ontology is: "*What are the syntactic representations of the tag* `title` *in a document dealing with movies?*" The expected answer, based on the set of real XML documents we used as our training set, is: "*The syntactic representations of the tag* `title` *are:* `movieTitle`, `titleMovie`, `t`, `Title`, `TITLE`." Figure 3 shows that all these representations are considered by the ontology. By checking the ontology, we realised that some motivating sce-

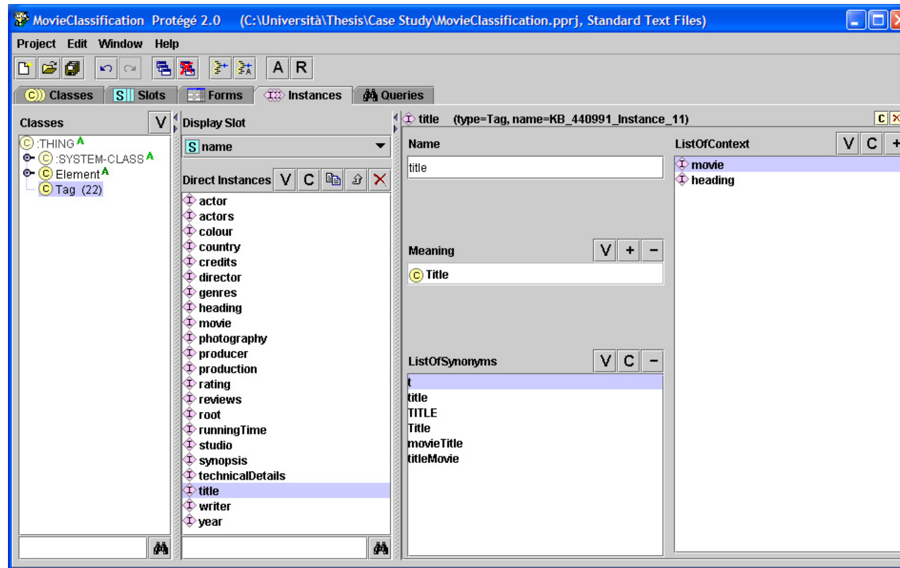


Fig. 3. One instance of the ontology

narios were not correctly addressed. In particular, the ontology did not include the information that `dirn` may be used as an alternative syntactic representation of the concept of `director`, which is indeed necessary to correctly classify the document in Table 3. Thanks to this testing, verification and validation stage we got feedback useful to refine the definition of the ontology.

– **Completeness check.**

  The last ontology development stage that we take under consideration in this paper is the completeness check. In order to face this stage, we restate all the informal competency questions as negative Horn clauses. Since we are going to use a Prolog interpreter to check whether or not these goals can be demonstrated starting from the Prolog program $Progr_{ontology}$ partly shown in Tables 4, 5, and 6, we take advantage of standard Prolog predicates to express conditions such as *X is not a variable* (`nonvar(X)`) and *X cannot be unified with Y* (`X \= Y`). An underscore (_) is used for unnamed variables for which no binding is required.

Below, we show how each competency question introduced in the "Requirement definition" stage can be expressed as a negative Horn clause Q_{Goal}, and which answer is computed by the Sicstus Prolog interpreter for $Progr_{ontology} \cup Q_{Goal}$. It is easy to see that the answers we got are consistent with the answers we expected, thus demonstrating the completeness of our ontology with respect to its requirements. By issuing a "`;`" command to the Sicstus Prolog interpreter after it returns one unification for the goal variables we force the interpreter to look for more answers. A `no` means that no more answers were found.

1. **Competency Question:** Should the ontology be able to separate the concepts related to the document structure from those related to the document semantics?
   Corresponding negative Horn clause:
   ```
   :- isA(tag, thing), isA(element, thing).
   ```
   Answer provided by the Sicstus Prolog interpreter:
   ```
   yes
   ```
2. **Competency Question:** What are the syntactic representations of tag "title" in the context of tag "heading"?
   Corresponding negative Horn clause:
   ```
   :- instanceOf(instance(title, _, _, SyntRepr), tag).
   ```
   Answer provided by the Sicstus Prolog interpreter:
   ```
   SyntRepr = [t, 'Title', title, 'TITLE' , movieTitle,
   titleMovie] ? ;
   SyntRepr = [t, 'Title', title, 'TITLE', movieTitle,
   titleMovie] ? ;
   no
   ```
   Here two answers are provided: one for the case the tag "title" is in the context of the tag "heading", and one for the case it is in the context of the tag "movie".
3. **Competency Question:** What is the meaning of the tag "title" in the context of the tag "heading"?
   Corresponding negative Horn clause:
   ```
   :- instanceOf(instance(title, heading, Meaning, _),
   tag).
   ```
   Answer provided by the Sicstus Prolog interpreter:
   ```
   Meaning = title ? ;
   no
   ```
4. **Competency Question:** Can the tag "actors" contain either a string or a list of "actor" tags?
   Corresponding negative Horn clause:
   ```
   :- instanceOf(instance(actor, actors, actor, _), tag),
   instanceOf(instance(actors, _, actor, _), tag).
   ```
   Answer provided by the Sicstus Prolog interpreter:
   ```
   yes
   ```
5. **Competency Question:** Can the information about the title of the film be an attribute of the tag "movie"?
   Corresponding negative Horn clause:
   ```
   :- instanceOf(instance(title, movie, _, _), tag).
   ```

Answer provided by the Sicstus Prolog interpreter:

`yes`

In this way we have demonstrated that our ontology is able to answer all the competency questions, moreover these answers are consistent with the XML documents retrieved from the web and used as motivating scenario during the development of the ontology.

5 Conclusions and future directions

In this paper we have outlined a methodology for developing ontologies which takes inspiration from three existing methodologies, namely OD101, UniMeth and EXPLODE. In particular, we have concentrated our efforts in the stage of checking the ontology completeness. Consistently with the existing literature on the topic [5,4], we suggest that the ontology developer performs the completeness check by formally defining the ontology as a set of definite Horn clauses (a Prolog program) and by stating the competency questions as negative Horn clauses (Prolog goals). The developer should then check that, for each competency question restated as a negative Horn clause, a refutation exists for the defined ontology and the competency question. From a practical point of view, this check can be carried out by means of any Prolog interpreter. We have used an example taken from the Semantic Web domain to illustrate our approach.

The main future direction of our work consists of the extension of the ontology for representing and retrieving XML documents in order to cope with other domains besides to "movie" one. The integration of such extended ontology into a prototypical peer-to-peer network implementing the ideas of [9] will demonstrate the suitability of our approach in a real scenario.

References

1. A. Castano, S. Ferrara, S. Montanelli, E. Pagani, and G. P. Rossi. Ontology-addressable contents in P2P networks. In *Proc. of the 1st SemPGRID Workshop*, 2003.
2. R. M. Colomb and R. Weber. Completeness and quality of an ontology for an information system. In N. Guarino, editor, *Proc. of FOIS'98*, pages 207–217. IOS-Press, 1998. `http://www.itee.uq.edu.au/~colomb/Papers/Ontology.html`.
3. V. Cordì, V. Mascardi, M. Martelli, and L. Sterling. Developing an ontology for the retrieval of xml documents: A comparative evaluation of existing methodologies. In *Proc. of AOIS'04*, 2004. `http://www.disi.unige.it/person/MascardiV/Download/CMMS04.pdf.gz`.
4. M. S. Fox and M. Gruninger. On ontologies and enterprise modelling. In *International Conference on Enterprise Integration Modelling Technology 97*. Springer-Verlag, 1997.
5. M. Gruninger and M. S. Fox. Methodology for the design and evaluation of ontologies. In *Proc. of the Workshop on Basic Ontological Issues in Knowledge Sharing*, 1995. `http://www.eil.utoronto.ca/enterprise-modelling/papers/gruninger-ijcai95.pdf`.
6. N. Guarino. Formal ontology, conceptual analysis and knowledge representation. *International Journal of Human Computer Studies*, 5/6(43):625–640, 1995.

7. M. Hristozova and L. Sterling. Experiences with ontology development for value-added publishing. In S. Cranefield, T. Finin, V. Tamma, and S. Willmott, editors, *Proc. of the OAS03 Workshop*, 2003. http://oas.otago.ac.nz/OAS2003/papers/OAS03-hristozova.pdf.
8. J. W. Lloyd. *Foundations of Logic Programming (2nd edition)*. Springer-Verlag, 1993.
9. M. Mesiti, G. Guerrini, and V. Mascardi. SAPORE P2P: A semantic, policy-based system for the retrieval of XML documents in P2P networks. Tech. rep., DISI-TR-04-05, Computer Science Department of Genova University. http://www.disi.unige.it/person/MascardiV/Download/MGM04.pdf.gz, 2004.
10. D. Milojicic, V. Kalogeraki, R. Lukose, K. Nagaraja, J. Pruyne, B. Richard, S. Rollins, and Z. Xu. Peer-to-peer computing. Tech. rep., HPL-2002-57, HP Labs Palo Alto., 2002.
11. W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Löser. Super-peer-based routing and clustering strategies for RDF-based peer-to-peer networks. In *Proc. of WWW'03*, pages 536–543, 2003.
12. N. F. Noy and D. L. McGuinness. Ontology development 101: A guide to creating your first ontology. Tech. rep., KSL-01-05, Stanford Knowledge Systems Laboratory., 2001.
13. The Knowledge Interchange Format Home Page. http://www-ksl.stanford.edu/knowledge-sharing/kif/.
14. M. Uschold. Building ontologies: Towards a unified methodology. In *Proc. of ES'96*, 1996. Also published as technical report, AIAI-TR-197. http://www.aiai.ed.ac.uk/project/ftp/documents/1996/96-es96-unified-method.ps.
15. M. Uschold and M. King. Towards a methodology for building ontologies. In *Proc. of the Workshop on Basic Ontological Issues in Knowledge Sharing*, 1995. Also published as technical report, AIAI-TR-183. http://www.aiai.ed.ac.uk/project/ftp/documents/1995/95-ont-ijcai95-ont-method.ps.

Combining logic programming and domain ontologies for text classification

Chiara Cumbo[2], Salvatore Iiritano[1], and Pasquale Rullo[2]

[1] Exeura s.r.l., `iiritano@exeura.it`
[2] Dipartimento di Matematica, Università della Calabria, 87030 Rende (CS), Italy, {`cumbo,rullo`}`@mat.unical.it`

Abstract This paper describes a prototypical system supporting the entire classification process: document storage and organization, pre-processing, ontology construction and classification. Document classification relies on two basic ideas: first, using ontologies for the formal representation of the domain knowledge; second, using a logic language (an extension of Datalog by aggregate functions that we call Datalog^f) as the categorization rule language. Classifying a document w.r.t. an ontology means associating it with one or more concepts of the ontology. Using Datalog^f provides the system with a natural and powerful tool for capturing the semantics provided by the domain ontology and describing complex patterns that are to be satisfied by (pre-processed) documents. The combined use of ontologies and Datalog^f allows us to perform a high-precision document classification.

1 Introduction

Managing the huge amount of textual documents available on the web and the intranets has become an important problem of knowledge management. For this reason, modern Knowledge Management Systems need for effective mechanisms to classify information and knowledge embedded in textual documents [1,2].
A number of classification approaches have been so far proposed, such as those based on machine learning [3] and those based on clustering techniques using the vector space model [4,5,6].
In this paper we describe a prototypical classification system which relies on two basic ideas: first, using ontologies for the formal representation of the domain knowledge and, second, using a logic language as the categorization rule language.
An ontology is a formal representation of an application domain [7,8]. In the context of a classification process, an ontology is intended to provide the specific knowledge concerning the universe of discourse (categorization based on the domain context). Classifying a document w.r.t. a given ontology means associating it with one or more concepts of the ontology. To this end, each concept is equipped with a set of logic rules that describe features of a document that may relate to the given concept. The logic language we use in our system is an extension of Datalog [9] with aggregate functions [10]. Throughout this paper

we refer to this language as Datalog^f. The advantage of using Datalog^f as the categorization rule language is twofold: first, we can exploit its expressive power to capture the domain semantics provided by the ontology and describe complex patterns that are to be satisfied by documents; second, the encoding of such patterns is very concise, simple, and elegant. We notice that others rule-based techniques have been proposed by several authors, but they are mainly devoted to the resolution of linguistic problems, such as the disambiguation of terms for the reduction of the vector dimensions [11], or for the improvement of the results of the classification task [3].
The execution of Datalog^f programs is carried out by the DLV system [12], which is part of our categorization engine. DLV is a well-known reasoning system which supports a completely declarative style of programming based on a bottom-up evaluation of the stable model semantics of disjunctive logic programs.

The paper is organized as follows. In Section 2 we provide an overview of the system. In Section 3 we describe the ontology management. In Section 4 we discuss the pre-processing phase and in Section 5 we present our classification technique based on Datalog^f. Finally, we give our conclusions.

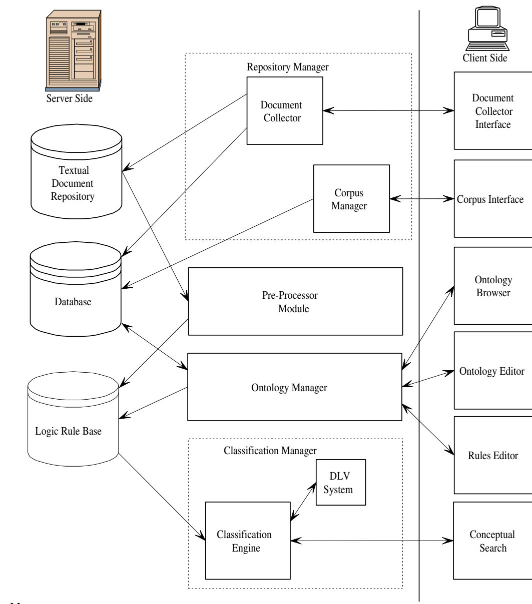
2 A system overview

The prototype is intended as a corporate classification system supporting the entire process life-cycle: document storage and organization, ontology construction, pre-processing and classification. It has been developed as a Web application based jsp-pages on the client side. A sketch of its architecture is shown in figure 1. In the following sections we shall focus our attention on ontology management, pre-processing and classification.

3 Ontology Management

Ontologies in our system provide the knowledge needed for a high-precision classification. The ontology specification language supports the following basic constructs: Concepts, Attributes, Properties (attribute values), Taxonomic (is-a and part-of) and Non-Taxonomic binary associations, Association cardinality constraints, Concept Instances, Links (association instances), Synonyms. The creation of an ontology is supported by the Ontology Editor which provides a powerful visual interface based on a graph representation.

Example 1. KIMOS is an ontology developed within Exeura (www.exeura.it) with the purpose of classifying all company's software resources and the respective documentation. A fragment of KIMOS is given in figure 2. Here, the central concept is "Software" which is related to the other concepts by both taxonomic and non-taxonomic relations. For an instance, the edge connecting "Software" with "Language" represents the (many-to-many) relation "developed-in", while the one between "Software" to "OS Compatible" represents the relation "runs-on"; the concept "Software" is subdivided into a number of sub-concepts that



..

Figure 1. The System Architecture

group the different instances of "Software" into the appropriate categories. In figure we have reported only the concept "DB" that represents the class of softwares for databases. This concept is related to "Software" by an is-a relation and it is classified into "DBMS" and "DB Tool". In turn, "DBMS" is classified as either "Relational DBMS" or "Others" (i.e., DBMS of different types). An instance of "Relational DBMS" is "MySQL" which is related to "Unix-C" (an instance of "OS Compatible") by the link "runs-on". □

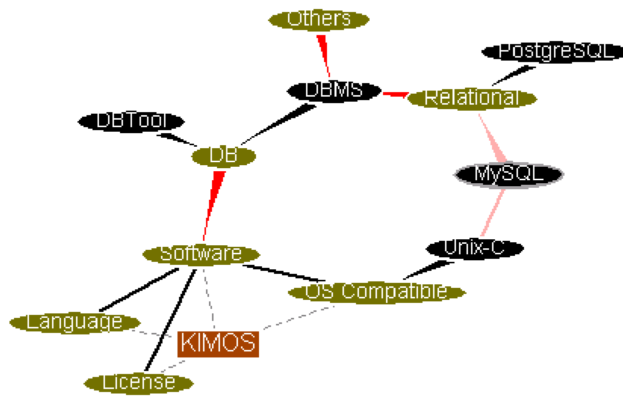
Once created, the user can navigate the ontology using the Ontology Browser which offers the following basic facilities:

- for a given concept, the user can easily explore its sub-concepts or, viceversa, collapse the underlying hierarchies
- the user can select the relationships whereby moving away from a given concept
- the user can filters the (possibly large) list of instances of a given concept.

Internally, an ontology is stored as a set of facts. As we will see in section 5, these facts represent an input to categorization programs.

Example 2. The internal representation of KIMOS consists of facts representing:

- concepts, e.g., *concept(DB)*;



..

Figure 2. The KIMOS Ontology

- attributes, each identified by an Id, a Data-Type and the Id of the concept which belongs to; for instance, *attribute(size-MB,real,Software)* represents the attribute "size-MB", of type real, of the concept "Software";
- Properties (i.e., attribute instances) each characterized by an attribute name and a value; for instance, *property(size-MB,1.35)*;
- Taxonomic relationships of the form *is-a(DB,Software)*;
- Non-taxonomic relationships such as *association(runs-on,Software,OS Compatible)* which represents the relation "runs-on" between "Software" and "OS Compatible"; we represent also the inverse *inverse-of(runs-on, supports)*;
- association cardinality constraints, e.g., *cardinality(runs-on, "$\geqslant$ 1")* and *cardinality(supports, "$\geqslant$ 1")*;
- Link associations (i.e., binary association between instances), e.g., *link(runs-on, MySql ,Unix-C)*;
- Concept Instances such as *instance-of(Relational DBMS,MySql)*;
- Synonyms such as *synonym(Database,DB)*.

□

4 Pre-processing

The aim of the Pre-Processing step is to obtain a machine-readable representation of textual documents [13]. This is done by *annotating* documents with meta-textual information obtained by a linguistic and structural analysis.
The Pre-Processor module supports the following tasks:

- Pre-Analysis, based on three main activities: Document Normalization, Structural Analysis and Tokenization.

- Linguistic Analysis, based on the following steps:
  - Lexical Analysis: for each token, the morpho-syntactical features are obtained (the stem and the Part of Speech - PoS). The PoS Tagging step is based on a variant of the Brill Tagger (a rule-based Pos-Tagger [14]).
  - Quantitative Analysis which provides, for a given document, information about the number of different tokens and stems as well as the absolute frequency for each token and stem.

The output of the Pre-Processing phase is a set of facts representing the relevant information about the processed document. As we shall see in section 5, these facts represent an input to our categorization programs.

Example 3. Consider, for example, a textual document about databases, with 247 different tokens, and suppose that the third paragraph of this document contains the following fragment of text: "... A **database** is a structured.... ". The representation of this paragraph is like this:

...
word(57,'a','a','at').
word(58,'database','databas','nn').
word(59,'is','is','bez').
word(60,'a','a','at').
word(61,'structured','structur','vbn').
bold(58).
par(3,57,148).
tokenFrequency('database',13).
stemFrequency('databas',16).
numberOfTokens(247).
numberOfStems(218). □

5 Document Classification

The basic idea is that of using logic programs to recognize concepts within texts. Logic rules, indeed, provide a natural and powerful way to describe features of document contents that may relate to concepts. To this end, we use the logic language Datalog^f [10], an extension of Datalog by aggregate functions. The module of our system which supports the classification process is the Classification Engine which relies on DLV system [12] for the bottom-up evaluation of Datalog^f programs.

5.1 The Datalog^f language

We call Datalog^f the logic language obtained by extending Datalog [9] by aggregate functions. A *function* has the form $f(Vars : Conj)$ where f is the name (count, sum, min, max, sum) and $Vars$ a set of variables occurring in the conjunction $Conj$. Intuitively the expression $Vars : Conj$ represents the

set of values assumed by the variables in $Vars$ making $Conj$ true. An *aggregate atom* is an expression of the type $Lg \leq f(Vars : Conj) \leq Ug$ where Lg and Ug are positive integer constants or variables called *guards*. For instance, $count\{V : a(V)\} < value$ is an aggregate atom whose informal meaning is: the number of ground instances of $a(V)$ must be less than $value$. A *Datalog^f program* is a logic program in which aggregate literals can occur in the body of rules. Rules with aggregate atoms are required to be $safe$ [10]. It is worth noticing that the result of an aggregate function can be saved by an assignment. For instance in the following rule $h(X) : -X = \#count\{V : a(V)\}$, all the ground instances of $a(V)$ are counted up and the value of count is assigned to X.

5.2 Categorization programs

By combining the expressive power of Datalog with that of aggregate functions, Datalog^f provides a natural and powerful tool for describing categorization rules within our system. A categorization program relies on a number of predefined predicates, that are of two types:

1. *Pre-processing predicates* representing information generated by the pre-processing phase; examples of such predicates are:
   $word(Id, Token, Stem, PoS)$
   $title(Id, Token, Stem, PoS)$
   where *Id* represents the position of *Token* within the text, *Stem* is the stem of the token and *PoS* its Part-of-Speech, and
   $tokenFrequency(Token, Number)$
   which represents the number of times Token occurs in the text.
2. *Ontology predicates* representing the domain ontology; examples of this kind of predicates are the following: $instance_of(I, C)$ (I is instance of the concept C), $synonym(C1, C2)$, $isa(C1, C2)$, $part_of(C1, C2)$, $association(A, C1, C2)$, etc..

In addition, we use the predicate $relevant(D, C)$ to state that document D is relevant for concept C.
Now, we equip each concept C of a given ontology with a set of Datalog^f rules, the *categorization program* P_C of C, used to recognize C within a given document D. The set of facts of P_C consists of the facts representing the domain ontology (see Section 3) as well as those representing the pre-processed document (see Section 4). The rules of P_C represent conditions that are to be satisfied in order D be considered relevant for C.

Example 4. We next provide an incremental construction of a categorization program associated with the concept "DB" of the KIMOS ontology (see example 1).

Rules looking for keyword. We start with the following simple rules looking for the keyword "DB":

r_0: $t0 : -title(_, "DB", _, _).$

r_1: $t1 : -tokenFrequency("DB", F), F > a.$

In rule r_0 above, the predicate $t0$ is true if "DB" occurs in the title, while t_1 in r_1 is true if the frequency F of the token "DB" is greater than a given constant a.
We can now refine our keyword search by exploiting synonyms; for instance, we can restate r_0 as

r_0: $t0 : -title(_, X, _, _), synonym(X, "DB").$

and replace r_1 by the following two rules:

r_2: $t2(X, F) : -synonym(X, "DB"), word(_, X, _, _),\ tokenFrequency(X, F).$

r_3: $t3$:- $F1 = \#sum\{F, X : t1(F, X)\}, F1 > a.$

Rule r_2 above "evaluates", for the concept "DB" and each of its synonyms, the respective frequency F; rule r_3, in turn, determines the total number $F1$ of times the concept "DB" and each of its synonyms appears in the text (this is performed by the aggregate function *sum*).

Rules looking for terms. Using the next rules we look for the term "structured data" within the document:

r_4 : $t4(I)$:- $word(I, "structured", _, _), word(J, "data", _, _), J = I + 1.$

r_5 : $t5(F)$:- $F = \#count\{I : t5(I)\}.$

We may relax the above condition, requiring the words "structured" and "data" to be found, in the specified order, within a distance of at most 5 words inside the same paragraph:

r_6 : $t6(I)$:- $word(I, "structured", _, _), word(J, "data", _, _), J > I,$
$L = J - I, L <= 5, sameParagraph(I, J).$

r_7: $sameParagraph(I, J)$:- $par(Id, Init, Fin), I >= Init, J <= Fin.$

r_8 : $t8(F)$:- $F = \#count\{I : t7(I)\}.$

Rule r_8 above counts the number of times the searched term occurs in the same paragraph.

Rules matching expressions. Next we write rules to recognize, within a para-

graph, an expression of the following type: a verb with stem "store", followed by a name having "tabl" or "relat" as its stem (i.e., we are trying to recognize sentences such as "data are stored within tables...").

r_9 : $t9(I)$:- $word(I, _, "store", "vb"), word(J, _, "tabl", _), sameParagraph(I, J)$.

r_{10} : $t10(I)$:- $word(I, _, "store", "vb"), word(J, _, "relat", _), sameParagraph(I, J)$.

r_{11} : $t11(F)$:- $F = \#count\{I : t9(I)\}$.

Rules exploiting the ontology knowledge. We can improve the precision of the classification process by using the underlying domain ontology. For instance, if a document talks about some specific instances of the concept "db", such as Oracle, Access, etc. (note that an instance of "relational DBMS", which is a sub-concept of "db", is also an instance of "DB"), it is quite obvious considering the document as pertinent to the concept "db". So, we write the following rules:

r_{12} : $t12(I, F)$:- $instance_of("DB", I), tokenFrequency(I, F)$.

r_{13}: $t13(N)$:- $N = \#count\{I : t11(I, _)\}$.

r_{14} : $t14(F)$:- $F = \#sum\{F1, I : t11(I, F1)\}$.

r_{15} : $t15(T)$:- $T = \#count\{I : instance_of(I, "DB")\}$.

where: r_{12} provides the number of occurrences of each instance of "db" in the document; r_{13} counts the number of distinct instances of "db"; r_{14} provides the total number of instances (duplicated included) of "db" and r_{15} gives the number of instances of "db" in the ontology. Finally, the rule

r_{16} : $t16(K, L)$:- $t13(N), t14(F), t15(T), K = N/T, L = F/N$.

expresses a measure, in terms of K (the fraction of the instances of "db" that are cited within the document) and L (which takes into account the fact that each instance might be cited several times), of the presence into the document of words representing instances of the concept "db". □

As we have mentioned before, we use DLV as the categorization engine in our system. DLV is a very powerful system for the bottom-up evaluation of disjunctive logic programs extended by a number of constructs (Datalog^f is a subset of the DLV language). It is used in many real applications where efficiency is a strong constraint.
The evaluation strategy of categorization programs is based on the following two observations:

- there are documents that are straightforward to classify, i.e., for which simple keyword-based rules (like $r_1 - r_2$ above) are enough; suppose, for instance,

that the word ”db” is contained in the title or it occurs frequently throughout the text; in such cases we can confidently classify the document at hand as relevant for the given concept only by using few simple rules (like r_1 and r_2) and forgetting of the remaining ones occurring in the rest of the categorization program;
- a deeper semantic analysis is needed only in case of documents that are difficult to classify because concepts do not appear explicitly; to this end, the execution of more complex rules (for instance, rules trying to match complex expressions) is required.

Now, the implementation of the above evaluation strategy proceeds, roughly speaking, as follows: we structure the categorization program P_C, associated to the concept C, into a number of components, say, $c_1, ..., c_n$. Each component groups rules performing some specific retrieval task, such as word-based search, term matching, etc., of increasing semantic complexity – that is, each component is capable to recognize texts that are possibly inaccessible to the ”previous” ones. Given a document D, the evaluation of P_C (w.r.t. D) starts from c_1 (the ”lowest” component) and, as soon as a component c_i, $1 \leq i \leq n$, is ”satisfied” (by D), the process stops successfully – i.e., D is recognized to be relevant for C and the fact $relevant(D, C)$ is stated to be true; if no such a component is found, the classification task fails.

5.3 Ontology-driven Classification Strategy

Let D be a document that has to be classified w.r.t. an ontology O. As we have seen in the previous subsection, each concept C of O is equipped with a suitable categorization program P_C whose evaluation determines whether D is relevant for C or not. An exhaustive approach would require to ”prove” D w.r.t. the categorization program of each concept of O, and this could result in a rather heavy computation. However, we can drastically reduce the ”search space” if we adopt an ontology-driven classification technique which exploits the presence of taxonomic hierarchies. This technique is based on the principle that if a document is relevant for a concept then it is so for all of its ancestors within an is-a taxonomy (unless the contrary is explicitly stated). This principle is expressed by the following recursive rule:

$$relevant(D, X) : -relevant(D, Y), isa(Y, X)$$

As an example, if a document is relevant for the concept ”Relational DBMS” of the KIMOS ontology, then it is so for the concepts ”DBMS”, ”DB” and ”Software”. If we want to exclude the latter, we simply write:

$$relevant(D, X) : -relevant(D, Y), isa(Y, X), X <> ''Software''.$$

The above inheritance principle suggests us a classification strategy where concepts within a sub-class hierarchy are processed in a bottom-up fashion. As soon

as D is found to be relevant for a concept C in the hierarchy H, it is not any more processed w.r.t. any of the ancestors of C in H. The relevance association of D to the ancestors of C is automatically performed by the above recursive rule.

6 Conclusion

We have presented a prototypical text classification system which relies on a combined use of ontologies and logic programming. The former are used to represent the domain knowledge, the latter to recognize concepts within texts. To this end, each concept of an ontology is equipped with a categorization program, i.e., a logic program written in Datalog^f – an extension of Datalog by aggregate functions. A categorization program is designed to discover complex patterns within texts using the knowledge provided by the underlying ontology. The classification process is ontology-driven and, as a result, provides a relationship between concepts and documents. The categorization engine is based on the logic programming system DLV.

So far, we have carried out a number of preliminary tests which seem to be very promising in terms of efficiency even on large documents. For an instance, we can classify a document of over 70000 words w.r.t. a Kimos Ontology (7 concepts) in 0.51 seconds. Further experimentation is currently being performed.

Current work is concerned with the extension of Datalog^f with external functions for the efficient execution of tasks such as stemming, substring matching, etc.

References

1. Ciravegna: (LP)2, an Adaptive Algorithm for Information Extraction from Web-related Texts. In: Proc. IJCAI-2001 Work. on Adaptive Text Extraction and Mining. (2001)
2. Riloff: A Case Study in Using Linguistic Phrases for Text Categorization on the WWW. In: AAAI/ICML Work.Learning for Text Categorization. (2001)
3. Cohen: Text categorization and relational learning. In: Proc. of ICML-95, 12th Int. Conference on Machine Learning. (1995)
4. Díaz, A., Buenaga, M., Urena, L., García-Vega, M.: Integrating linguistic resources in an uniform way for text classification tasks. In: Proc. of LREC-98, 1st Int. Conference on Language Resources and Evaluation. (1998) 1197–1204
5. Hsu: Classification algorithms for NETNEWS articles. In: Proc. of CIKM-99, 8th ACM Int. Conference on Information and Knowledge Management. (1999) 114–121
6. Brank, J., Grobelnik, M., Milic-Frayling, N., Mladenic, D.: Feature selection using support vector machines. In: Proc. of the 3rd International Conference on Data Mining Methods and Databases for Engineering, Finance, and Other Fields. (2002)
7. Decker, S., Erdmann, M., Fensel, D., Studer, R. In: Ontobroker: Ontology Based Access to Distributed and Semi-Structured Information. Proc. of DS-8. Kluwer Academic Publ (1999) 351–369
8. Fensel: OIL: An ontology infrastructure for the semantic web. IEIS **16** (2001) 38–45

9. Ullman: Principles of Database and Knowledge-Base Systems, Rockville (Md.) (1988)
10. Dell'Armi, T., Faber, W., Ielpa, G., Leone, N., Pfeifer, G.: Aggregate Functions in Disjunctive Logic Programming: Semantics, Complexity, and Implementation in DLV. In: Proc. IJCAI 2003, Acapulco, Mexico, Morgan Kaufmann Publishers (2003)
11. Paliouras: Learning rules for large vocabulary word sense disambiguation. In: Proc. of IJCAI-99. (1999) 674–679
12. Faber, W., Pfeifer, G.: DLV homepage (since 1996) http://www.dlvsystem.com/.
13. Yang: A comparative study on feature selection in text categorization. In: International Conference on Machine Learning, ACL (1997) 412–420
14. Brill: Tranformation-based error-driven learning and natural language processing: A case study in part of speech tagging. In: Computational Linguistics. (1995) 543–565

Ontological encapsulation of many-valued logic

Zoran Majkić

Dipartimento di Informatica e Sistemistica, University of Roma “La Sapienza”
Via Salaria 113, I-00198 Rome, Italy
majkic@dis.uniroma1.it
http://www.dis.uniroma1.it/ $\sim$ majkic/

Abstract. Large databases obtained by the data integration of different source databases can be incomplete and inconsistent in many ways. The classical logic is not the appropriate formalism for reasoning about inconsistent databases. Certain local inconsistencies should not be allowed to significantly alter the intended meaning of such logic programs. The variety of semantical approaches that have been invented for logic programs is quite broad. In particular we are interested for many-valued logics with negation, based on bilattices. We present a 2-valued logic, based on an Ontological Encapsulation of Many-Valued Logic Programming, which overcome some drawbacks of the previous research approaches in many-valued logic programming. We defined a Model theory for Herbrand interpretations of ontologically encapsulated logic programs, based on a semantic reflection of the epistemic many-valued logic.

1 Introduction to Many-valued logic programming

Semantics of logic programs are generally based on a classical 2-valued logic by means of stable models, [1,2]. Under these circumstances not every program has a stable model. Three-valued, or partial model semantics had an extensive development for logic programs generally, [3,4]. Przymusinski extended the notion of stable model to allow 3-valued, or partial, stable models, [5], and showed every program has at least one partial stable model, and the well-founded model is the smallest among them, [6]. Once one has made the transition from classical to partial models allowing *incomplete* information, it is a small step to also allow models admitting *inconsistent* information. Doing so provides a natural framework for the semantic understanding of logic programs that are distributed over several sites, with possibly conflicting information coming from different places. As classical logic semantics decrees that inconsistent theories have no models, classical logic is not the appropriate formalism for reasoning about inconsistent databases: certain ”localizable” inconsistences should not be allowed to significantly alter the intended meaning of such databases.
So far, research in many-valued logic programming has proceeded along different directions: *Signed* logics [7,8] and *Annotated* logic programming [9,10] which can be embedded into the first, *Bilattice-based* logics, [11,12], and *Quantitative rule-sets*, [13,14]. Earlier studies of these approaches quickly identified various distinctions between these frameworks. For example, one of the key insights behind bilattices was the interplay between the truth values assigned to sentences and the (non classic) notion of *implication* in the language under considerations. Thus, rules (implications) had

weights (or truth values) associated with them as a whole. The problem was to study how truth values should be propagated "across" implications. Annotated logics, on the other hand, appeared to associate truth values with each component of an implication rather than the implication as a whole. Roughly, based on the way in which uncertainty is associated with facts and rules of a program, these frameworks can be classified into *implication based* (IB) and *annotation based* (AB).
In the IB approach a rule is of the form $A \leftarrow^{\alpha} B_1, .., B_n$, which says that the certainty associated with the implication is α. Computationally, given an assignment I of logical values to the B_is, the logical value of A is computed by taking the "conjunction" of logical values $I(B_i)$ and then somehow "propagating" it to the rule head A.
In the AB approach a rule is of the form $A : f(\beta_1, .., \beta_n) \leftarrow B_1 : \beta_1, ..., B_n : \beta_n$, which asserts "the certainty of the atom A is least (or is in) $f(\beta_1, .., \beta_n)$, whenever the certainty of the atom B_i is at least (or is in) β_i, $1 \leq i \leq n$", where f is an n-ary computable function and β_i is either constant or a variable ranging over many-valued logic values.
The comparison in [15] shows:
1- while the way implication is treated on the AB approach is closer to the classical logic, the way rules are fired in the IB approach has definite intuitive appeal.
2- the AB approach is strictly more expressive than IB. The down side is that query processing in the AB approach is more complicated, e.g. the fixpoint operator is not continuous in general, while it is in the IB approaches.
3- the Fitting fixpoint semantics for logic programs, based exclusively on a bilattice-algebra operators, suffer two drawbacks: the lack of the notion of tautology (bilattice negation operator is an *epistemic* negation) leads to difficulties in defining proof procedures and to the need for additional complex truth-related notions as "formula closure"; there is an unpleasant asymmetry in the semantics of implication (which is strictly 2-valued) w.r.t. all other bilattice operators (which produce any truth value from the bilattice) - it is a sign that strict bilattice language is not enough expressive for logic programming, and we need some reacher (different) syntax for logical programming.
From the above points, it is believed that IB approach is easier to use and is more amenable for efficient implementations, but also annotated syntax (but with IB semantics) is useful to overcome two drawbacks above: the syntax of new encapsulated many-valued logic (in some sense 'meta'-logic for a many-valued bilattice logic) will be 2-valued and can be syntactically seen as a kind of very simple annotated syntax. Thus the implication (and classical negation also), not present in a bilattice algebra operators, will have a natural semantic interpretation in this enriched framework.
In [10] it is shown how the Fitting's 3-valued bilattice logic can be embedded into an Annotated Logic Programming which is computationaly very complex. The aim of this work is (1) to extend the Fittihg's fixpoint semantics to deal with inconsistencies also, and (2) to define the notion of a model for such many-valued logic programs by some kind of 'minimal' (more simple and less computationally expensive than APC) logic. In order to respond to these questions we (1) introduce *built-in predicates* in the heads of clauses, and (2) *encapsulate* the 'object' epistemic many-valued logic programs into 2-valued 'meta' ontological logic programs. We argue that such logic will be good framework for supporting the data integration systems with key and foreign

key integrity constraints with incomplete and inconsistent source databases, with less computation complexity for certain answers to conjunctive queries [16,17].
The plan of this paper is the following: Section 2 introduce the Belnap's bilattice concepts and the particular 4-valued version, $\mathcal{B}_4$, used in this paper. In Section 3 is presented an inference framework for a 4-valued bilattice based logic, particularly for derivation of *possible* facts (w.r.t. true and false facts as in 3-valued strong Kleene's logic) and is given a representation theorem for this 4-valued logic. In Section 4 is developed conceptual framework for encapsulation of this epistemic 'object' 4-valued logic into an ontological 'meta' 2-valued logic by mean of *semantic reflection*. Moreover, is given the definition for a 4-valued implication useful for inconsistent databases and an example where inconsistency is managed by clauses with built-in predicate in a head. Finally, Section 5 defines the syntax and the *model theoretic* Herbrand semantics for the ontological encapsulation of many-valued logic programs.

2 Many-valued epistemic logic based on a Bilattice

In [18], Belnap introduced a logic intended to deal in a useful way with inconsistent or incomplete information. It is the simplest example of a non-trivial bilattice and it illustrates many of the basic ideas concerning them. We denote the four values as $\{t, f, \top, \bot\}$, where t is *true*, f is *false*, $\top$ is inconsistent (both true and false) or *possible*, and $\bot$ is *unknown*. As Belnap observed, these values can be given two natural orders: *truth* order, $\leq_t$, and *knowledge* order, $\leq_k$, such that $f \leq_t \top \leq_t t$, $f \leq_t \bot \leq_t t$, and $\bot \leq_k f \leq_k \top$, $\bot \leq_k t \leq_k \top$. This two orderings define corresponding equivalences $=_t$ and $=_k$. Thus any two members α, β in a bilattice are equal, $\alpha = \beta$, if and only if (shortly 'iff') $\alpha =_t \beta$ and $\alpha =_k \beta$.
Meet and join operators under $\leq_t$ are denoted $\wedge$ and $\vee$; they are natural generalizations of the usual conjunction and disjunction notions. Meet and join under $\leq_k$ are denoted $\otimes$ (*consensus*, because it produces the most information that two truth values can agree on) and $\oplus$ (*gullibility*, it accepts anything it's told), such that hold:
$f \otimes t = \bot$, $f \oplus t = \top$, $\top \wedge \bot = f$ and $\top \vee \bot = t$.
There is a natural notion of truth negation, denoted $\sim$, (reverses the $\leq_t$ ordering, while preserving the $\leq_k$ ordering): switching f and t, leaving $\bot$ and $\top$, and corresponding knowledge negation, denoted $-$ (reverses the $\leq_k$ ordering, while preserving the $\leq_t$ ordering), switching $\bot$ and $\top$, leaving f and t. These two kind of negation commute: $- \sim x = \sim -x$ for every member x of a bilattice.
It turns out that the operations $\wedge, \vee$ and $\sim$, restricted to $\{f, t, \bot\}$ are exactly those of Kleene's strong 3-valued logic. Any bilattice $\langle \mathcal{B}, \leq_t, \leq_k \rangle$ is:
1. *Interlaced*, if each of the operations $\wedge, \vee, \otimes$ and $\oplus$ is monotone with respect to both orderings (for instance, $x \leq_t y$ implies $x \otimes z \leq_t y \otimes z$, $x \leq_k y$ implies $x \wedge z \leq_k y \wedge z$).
2. *Infinitarily interlaced*, if it is complete and four infinitary meet and join operations are monotone with respect to both orderings.
3. *Distributive*, if all 12 distributive laws connecting $\wedge, \vee, \otimes$ and $\oplus$ are valid.
4. *Infinitarily distributive*, if it is complete and infinitary, as well as finitary, distributive laws are valid. (Note that a bilattice is *complete* if all meets and joins exist, w.r.t. both orderings. We denote infinitary meet and join w.r.t. $\leq_t$ by $\bigwedge$ and $\bigvee$, and by $\prod$ and $\sum$

for the $\leq_k$ ordering; for example, the distributive law for $\otimes$ and $\bigwedge$ may be given by $x \otimes \bigwedge_i y_i = \bigwedge_i (x \otimes y_i)$).
A more general information about bilattice may be found in [19]: he also defines *exact* members of a bilattice, when $x = -x$ (they are 2-valued consistent), and *consistent* members, when $x \leq_k -x$ (they are 3-valued consistent), but a specific 4-valued consistence will be analyzed in the following paragraphs.
The Belnap's 4-valued bilattice is infinitary distributive. In the rest of this paper we denote by $\mathcal{B}_4$ a special case of the Belnap's bilattice. In this way we consider the *possible* value as weak true value and not as inconsistent (that is true and false together). We have more knowledge for ground atom with such value, w.r.t. the true ground atom, because we know also that if we assign the true value to such atom we may obtain an inconsistent database.

3 Representation theorem

Ginsberg [11] defined a world-based bilattices, considering a collection of worlds W, where by world we mean some possible way of things might be, and where $[U, V]$ is a pair of subsets of W which express truth of some sentence p, with $\leq_t, \leq_k$ truth and knowledge preorders relatively, as follows:
1. U is a set of worlds where p is true, V is a set of worlds where p is false, $P = U \bigcap V$ is a set where p is inconsistent (both true and false), and $W - (U \bigcup V)$ is a set where p is unknown.
2. $[U, V] \leq_t [U_1, V_1]$ iff $U \subseteq U_1$ and $V_1 \subseteq V$
3. $[U, V] \leq_k [U_1, V_1]$ iff $U \subseteq U_1$ and $V \subseteq V_1$
Such definition is well suited for the 3-valued Kleene logic, but for the 4-valued logic used to overcome "localizable" inconsistencies it is not useful, mainly for two following reasons:
1. The *inconsistent* (both true and false) top knowledge value in the Belnap's bilattice can't be assigned to sentences, otherwise we will obtain an inconsistent logic theory; because of that consistent logics in this interpretation can have only three remaining values. Thus we interpret it as *possible* value, which will be assigned to mutually inconsistent sentences, and we obtain possibility to have consistent 4-valued logic theories in order to overcome such inconsistencies.
2. Let denote by $T = U - P$, $F = V - P$, where P is a set of worlds where p has a possible logic value. Then we obtain that $[U, V] \leq_t [U_1, V_1]$ also when $T \supset T_1$, which is in contrast with our intuition. Consequently, we adopt a triple $[T, P, F]$ of mutually disjoint subsets of W to express truth of some sentence p ($W - T \bigcup P \bigcup F$ are worlds where p is unknown), with the following definition for their truth and knowledge orders:
2.1 $[T, P, F] \leq_t [T_1, P_1, F_1]$ iff $T \subseteq T_1$ and $F_1 \subseteq F$
2.2 $[T, P, F] \leq_k [T_1, P_1, F_1]$ iff $T \subseteq T_1$, $P \subseteq P_1$ and $F \subseteq F_1$.
Let us try now to render *more rational* these two intuitions described above. In order to obtain a new bilattice abstraction rationality, useful to manage logic programs with possible 'localizable' inconsistencies, we need to consider more deeply the *fundamental phenomena* in such one framework. In the process of derivation of new facts, for a given logic program, based on the 'immediate consequence operator', we have the following

three truth transformations for ground atoms in a Herbrand base of such program:
1. When ground atom pass from *unknown* to *true* logic value, without generating inconsistence. Let denote this action by $\uparrow_1 : \perp \rightarrowtail t$. The preorder of this 2-valued sublattice of $\mathcal{B}$, $L_1 = \{\perp, t\}$, defined by the direction of this transformation, 'truth increasing', is $\leq_1 \equiv \leq_t$. The meet and join operators for this lattice are $\wedge, \vee$ respectively. It is also knowledge increasing.
2. When some ground atom, try to pass from unknown to true/false value, generating an inconsistency, then is applied the *inconsistency repairing*, that is the *true* value of the literal of this atom, in a body of a violated clause with built-in predicate, is replaced by *possible* value. Let denote this action by $\uparrow_2 : t \rightarrowtail \top$. The preorder of this 2-valued sublattice of $\mathcal{B}$, $L_2 = \{t, \top\}$, defined by the direction of this transformation, 'knowledge increasing'. The meet and join operators for this lattice, w.r.t. this ordering are $\otimes, \oplus$ respectively. Notice that this transformation *does not change* the truth ordering because the ground atom pass from unknown to possible value.
3. When ground atom pass from *unknown* to *false* logic value, without generating inconsistence. Let denote this action by $\uparrow_3 : \perp \rightarrowtail f$. The preorder of this 2-valued sublattice of $\mathcal{B}$, $L_3 = \{\perp, f\}$, defined by the direction of this transformation, 'falsehood increasing' (inverse of 'truth increasing'), is $\leq_3 \equiv \leq_t^{-1}$. The meet and join operators for this lattice are $\vee, \wedge$ respectively. It is also knowledge increasing.
Thus, any truth transformation in some multi-valued logic theory (program) can be seen as composition of these three orthogonal dimensional transformations,i.e. by triples (or *multi-actions*), $[a_1, a_2, a_3]$, acting on the idle (default) state $[\perp, t, \perp]$; for instance the multi-action $[_, _, \uparrow_3]$, composed by the singe action $\uparrow_3$, applied to the default state generates the "false" state $[\perp, t, f]$. The default state $[\perp, t, \perp]$ in this 3-dimensional space has role as unknown value for single-dimensional bilattice transformations, that is it is a "unknown" state. Consequently, we define this space of states by the cartesian product of single-dimensional lattices, $L_1 \times L_2 \times L_3$, composed by triples $[x, y, z]$, $x \in L_1 = \{\perp, t\}$, $y \in L_2 = \{t, \top\}$ and $z \in L_3 = \{\perp, f\}$.

Definition 1. *By $L_1 \odot L_2 \odot L_3$ we mean the bilattice $< L_1 \times L_2 \times L_3, \leq_t^B, \leq_k^B >$ where, given any $X = [x, y, z]$, and $X_1 = [x_1, y_1, z_1]$:*
1. Considering that the second transformation does not influence the truth ordering,
$X \leq_t^B X_1$ *if* $x \leq_1 x_1$ *and* $z \leq_3 z_1$ *, i.e., if* $x \leq_t x_1$ *and* $z \geq_t z_1$
2. Considering that all three transformations are knowledge increasing, we have
$X \leq_k^B X_1$ *if* $x \leq_k x_1$ *and* $y \leq_k y_1$ *and* $z \leq_k z_1$
3. $X \wedge_B X_1 =_{def} [(x \wedge_1 x_1,\ y \wedge_1 y_1), z \wedge_3 z_1] = [x \wedge x_1,\ y \wedge y_1,\ z \vee z_1]$
4. $X \vee_B X_1 =_{def} [x \vee_1 x_1, (y \vee_3 y_1,\ z \vee_3 z_1)] = [x \vee x_1,\ y \wedge y_1,\ z \wedge z_1]$
5. $X \otimes_B X_1 =_{def} [x \otimes x_1,\ y \otimes y_1,\ z \otimes z_1]$
6. $X \oplus_B X_1 =_{def} [x \oplus x_1,\ y \oplus y_1,\ z \oplus z_1]$

These three bilattice transformations can be formally defined by lattice homomorphisms.

Proposition 1 *The following three lattice homomorphisms defines the 3-dimensional truth transformations:*
1. Truth dimension, $\theta_1 = _ \vee \perp$: $(\mathcal{B}, \wedge, \vee, \otimes, \oplus) \rightarrow (L_1, \wedge_1, \vee_1, \otimes, \oplus)$,
with $\wedge_1 = \wedge$, $\vee_1 = \vee$. *This is a strong positive transformation, which transforms*

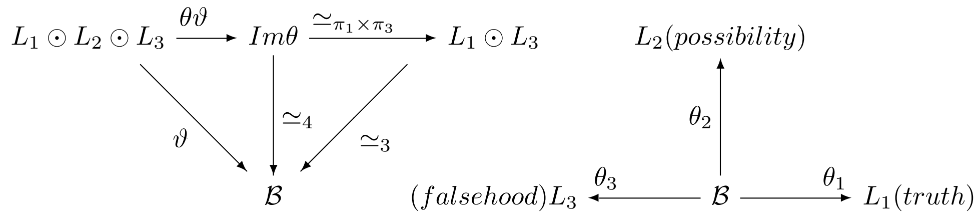
falsehood into unknown and possibility in truth.
2. Possibility dimension, $\theta_2 = _\vee \sim _ \vee \top : (\mathcal{B}, \otimes, \oplus) \rightarrow (L_2, \otimes, \oplus)$. *This is a weak knowledge transformation which transform unknown into possibility.*
3. Falsehood dimension, $\theta_3 = _\wedge \perp : (\mathcal{B}, \vee, \wedge, \otimes, \oplus) \rightarrow (L_3, \wedge_3, \vee_3, \otimes, \oplus)$,
with $\wedge_3 = \vee$, $\vee_3 = \wedge$. *This is a strong negative transformation, which transforms truth into unknown and possibility into falsehood.*
We define the following two mappings between Belnap's and its derived bilattice:
Dimensional partitioning: $\theta =< \theta_1, \theta_2, \theta_3 >: \mathcal{B} \rightarrow L_1 \odot L_2 \odot L_3$ *and*
Collapsing: $\vartheta : L_1 \odot L_2 \odot L_3 \rightarrow \mathcal{B}$, *such that* $\vartheta(x_1, x_2, x_3) =_{def} (x_1 \oplus x_3) \wedge x_2$.

These three lattice homomorphisms preserves the bilattice structure of $\mathcal{B}$ into the space of states $L_1 \odot L_2 \odot L_3$. That is we have that ($'_'$ represents no action)
$\theta(\perp) = [_, _, _]([\perp, t, \perp]) = [\perp, t, \perp]$, unknown state
$\theta(f) = [_, _, \uparrow_3]([\perp, t, \perp]) = [\perp, t, f]$, false state
$\theta(t) = [\uparrow_1, _, _]([\perp, t, \perp]) = [t, t, \perp]$, true state
$\theta(\top) = [\uparrow_1, \uparrow_2, \uparrow_3]([\perp, t, \perp]) = [t, \top, f]$, possible state.
Notice that the multi-action $[\uparrow_1, \uparrow_2, \uparrow_3]$ represents two cases for repairing inconsistencies: first, when unknown value of some ground atom tries to become true (action $\uparrow_1$) but makes inconsistency, thus is applied also action $\uparrow_2$ to transform it into possible value; second, when unknown value of some ground atom tries to become false (action $\uparrow_3$) but makes inconsistency, thus is applied also action $\uparrow_2$ to transform it into possible value. Notice that the isomorphism between the set of states and the set of multi-actions $\{[a_1, a_2, a_3] \mid a_1 \in \{\uparrow_1, _\},\ a_2 \in \{\uparrow_2, _\},\ a_3 \in \{\uparrow_3, _\}\}$ defines the *semantics* to the bilattice $L_1 \odot L_2 \odot L_3$.

Proposition 2 *Let* $Im\theta \subseteq L_1 \odot L_2 \odot L_3$ *be the bilattice obtained by image of Dimensional partitioning. It has also unary operators:*
Negation, $\sim_B = \theta \sim \vartheta$ *, and conflation,* $-_B = \theta - \vartheta$.

It is easy to verify that $\vartheta \circ \theta = id_{\mathcal{B}}$ is an identity on $\mathcal{B}$, and that ϑ is surjective with $\theta \circ \vartheta = id_{Im\theta}$. The negation $\sim_B$ preserves knowledge and inverts truth ordering and $\sim_B \sim_B X = X$; the conflation $-_B$ preserves truth and inverts knowledge ordering and $-_B -_B X = X$; and holds the commutativity $\sim_B -_B = -_B \sim_B$. (for example, $\sim_B -_B = \theta \sim \vartheta\theta - \vartheta = \theta \sim id_{\mathcal{B}} - \vartheta = \theta \sim -\vartheta = \theta - \sim \vartheta = \theta - \vartheta\theta \sim \vartheta = -_B \sim_B$).
So, we obtain that, for any $X = [x, y, z]$, hold $\sim_B X =_{def} [\sim z,\ y, \sim x]$ and $-_B X =_{def} [\theta_1(-z),\ \theta_2(-\vartheta(X)),\ \theta_3(-x)]$.

Theorem 1. *(Representation theorem) If* $\mathcal{B}$ *is a 4-valued distributive lattice then there are its distributive sublattices,* L_1, L_2, L_3*, such that* $\mathcal{B}$ *is isomorphic to the sublattice of* $L_1 \odot L_2 \odot L_3$ *defined by image of Dimensional partitioning* $Im\theta$ *. Moreover the following diagram (on the left) of bilattice homomorphisms commute*



where $\simeq_{\pi_1 \times \pi_3}$ is a projection isomorphism, $\simeq_3$ is the isomorphism (restriction of ϑ to the projection $L_1 \odot L_3$) of Fitting's representation Th. [20] valid for a 3-valued logics, and $\simeq_4$ is new 4-valued isomorphism (restriction of ϑ to $Im\theta$, and inverse to θ).
If $\mathcal{B}$ has negation and conflation operators that commute with each other, they are preserved by all isomorphisms of the right commutative triangle.

Proof. It is easy to verify that all arrows are homomorphisms (w.r.t. binary bilattice operators). The following table represents the correspondence of elements of these bilattices defined by homomorphisms:

$Multi-actions$	$L_1 \odot L_2 \odot L_3$	$Im\theta$	$L_1 \odot L_3$	$\mathcal{B}$
$[_, _, _]$	$[\bot, t, \bot]$	$[\bot, t, \bot]$	$[\bot, \bot]$	$\bot$
$[_, \uparrow_2, _]$	$[\bot, \top, \bot]$			
$[_, _, \uparrow_3]$	$[\bot, t, f]$	$[\bot, t, f]$	$[\bot, f]$	f
$[_, \uparrow_2, \uparrow_3]$	$[\bot, \top, f]$			
$[\uparrow_1, _, _]$	$[t, t, \bot]$	$[t, t, \bot]$	$[t, \bot]$	t
$[\uparrow_1, \uparrow_2, _]$	$[t, \top, \bot]$			
$[\uparrow_1, \uparrow_2, \uparrow_3]$	$[t, \top, f]$	$[t, \top, f]$	$[t, f]$	$\top$
$[\uparrow_1, _, \uparrow_3]$	$[t, t, f]$			

Let prove, for example, that the isomorphism $\theta : \mathcal{B} \rightarrow Im\theta$ preserves negation and conflation: $\sim_B \theta(x) = \theta \sim \vartheta\theta(x) = \theta \sim id_B(x) = \theta(\sim x)$, and
$-_B\theta(x) = \theta - \vartheta\theta(x) = \theta - id_B(x) = \theta(-x)$.

4 Semantic reflection of the epistemic logic

We assume that the Herbrand universe is $\Gamma_U = \Gamma \bigcup \Omega$, where Γ is ordinary domain of database constants, and Ω is an infinite enumerable set of marked null values, $\Omega = \{\omega_0, \omega_1, ...\}$, and for a given logic program P composed by a set of predicate and function symbols, P_S, F_S respectively, we define a set of all terms, $\mathcal{T}_S$, and its subset of ground terms $\mathcal{T}_0$, then atoms are defined as:
$\mathcal{A}_S = \{p(c_1, .., c_n) \mid p \in P_S,\ n = arity(p)\ and\ c_i \in \mathcal{T}_S\}$
The Herbrand base, H_P, is the set of all ground (i.e., variable free) atoms. A (ordinary) Herbrand interpretation is a many-valued mapping $I : H_P \rightarrow \mathcal{B}$. If P is a many-valued logic program with the Herbrand base H_P , then the ordering relations and operations in a bilattice $\mathcal{B}_4$ are propagated to the function space $\mathcal{B}_4^{H_P}$, that is the set of all Herbrand interpretations (functions), $I = v_B : H_P \rightarrow \mathcal{B}_4$, as follows:

Definition 2. *Ordering relations are defined on the Function space $\mathcal{B}_4^{H_P}$ pointwise, as follows: for any two Herbrand interpretations $v_B, w_B \in \mathcal{B}_4^{H_P}$*
1. $v_B \leq_t w_B$ if $v_B(A) \leq_t w_B(A)$ for all $A \in H_P$.
2. $v_B \leq_k w_B$ if $v_B(A) \leq_k w_B(A)$ for all $A \in H_P$.
3. $\sim v_B$ is the interpretation such that $(\sim v_B)(A) =\sim (v_B(A))$.
4. $-v_B$ is the interpretation such that $(-v_B)(A) = -(v_B(A))$.

It is straightforward [19] that this makes a function space $\mathcal{B}_4^{H_P}$ itself a complete infinitary distributive bilattice.

One of the key insights behind bilattices [11,12] was the interplay between the truth values assigned to sentences and the (non classic) notion of *implication*. The problem was to study how truth values should be propagated ”across” implications. In [21] is proposed the following IB based approach to the ’object’ 4-valued logic programming, which extends the definition given for a 3-valued logic programming [5]:

Definition 3. *Let $\mathcal{P}_B$ be the set of built-in predicates. The valuation, $v_B : H_P \rightarrow \mathcal{B}_4$, is extended to logic implication of a ground clause $p(\mathbf{c}) \leftarrow B$, where $B = B_1 \wedge .. \wedge B_n$, as follows:*

$$v_B(B \rightarrow p(\mathbf{c})) = t\,, \quad iff \qquad v_B(p(\mathbf{c})) \geq_t v_B(B) \; or \; (v_B(B) = \top \; and \, p \in \mathcal{P}_B)$$

Inconsistency acceptance: if $p \in \mathcal{P}_B$ is a built-in predicate, this clause is satisfied also when $v_B(p(\mathbf{c})) = f$ and $v_B(B) = \top$. This principle extends the previous definition of implication based only on truth ordering.
In order to obtain such many-valued definition, which generalize the 2-valued definition given above we will consider the conservative extensions of Lukasiewicz’s and Kleene’s strong 3-valued matrices (where third logic value $\perp$ is considered as unknown). So we obtain the following matrix, $f_{\leftarrow} : \mathcal{B} \times \mathcal{B} \rightarrow \mathcal{B}$, for implication ($\alpha = t$ and $\alpha = \perp$ for Lukasiewicz’s and Kleene’s case, respectively):

$\rightarrow$	t	$\perp$	$\top$	f
t	t	$\perp$	$\top$	f
$\perp$	t	α	$\top$	$\perp$
$\top$	t	t	t	t
f	t	t	t	t

For our purpose we assume the Lukasiewicz’s extension, i.e. $\alpha = t$, in order to have a tautology $a \leftarrow a$ for any formula a, and also to guarantee the truth of a clause (implication) $p(\mathbf{c}) \leftarrow B$, whenever $v_B(p(\mathbf{c})) \geq_t v_B(B)$, as used in fixpoint semantics for ’immediate consequence operators’. Such conservative extensions are based on the following observation: the problem to study how the truth values should be propagated ”across” implications can be restricted only to *true* implications (in fact we don’t use implications when are not true, because the ’immediate consequence operator’ derives new facts only for *true* clauses, i.e. when implication is true).
Example 1: The *built-in predicates* (ex, $=, \leq, \geq, ..$) may be used for integrity constraints: let $p(x, y)$ be a predicate and we define the key-constraint for attributes in x by $(y = z) \leftarrow p(x, y), p(x, z)$, where the atom $y = z$ is based on the built-in predicate $'='$. Let consider a program : $\quad p(x, y) \leftarrow r(x, y)\,, \quad (y = z) \leftarrow p(x, y),\ p(x, z)$
where r is a source database relation with two tuples, $(a, b),\ (a, c)$, p is a virtual relation of this database with key constraint, and x, y, z are object variables. The built-in predicates have the same prefixed extension in *all* models of a logic program, and that their ground atoms are *true or false*. If we assume that, $r(a, b),\ r(a, c)$ are true, then such facts are mutually inconsistent for p because of key constraint ($b = c$ is false). Thus, only one of them may be true in any model of this logic program, for example $r(a, b)$. So, if we assign the ’possible’ value $\top$ to $r(a, c)$ (or to both of them), we obtain that the clause $(b = c) \leftarrow p(a, b),\ p(a, c)$, thanks to the *inconsistency acceptance*, is satisfied.

Each *Herbrand interpretation* is a valuation. Valuations can be extended to maps from the set of all ground (variable free) formulas to $\mathcal{B}$ in the following way:

Definition 4. *Let $\mathcal{P}_S$ be the set of all predicate symbols ($\mathcal{P}_B \subseteq \mathcal{P}_S$ is a subset of built-in predicates), $\boldsymbol{e}$ the special (error) singleton, and $I : H_P \rightarrow \mathcal{B}$ be a many-valued Herbrand interpretation. A valuation I determines:*
1. A Generalized interpretation mapping $\mathcal{I} : \mathcal{P}_S \times \bigcup_{i \leq \omega} \mathcal{T}_0^i \rightarrow \mathcal{B} \bigcup \{\boldsymbol{e}\}$, such that for any $\boldsymbol{c} = (c_1, .., c_n) \in \mathcal{T}_0^n$, $\mathcal{I}(p, \boldsymbol{c}) = I(p(\boldsymbol{c}))$ iff $arity(p) = n$; $\boldsymbol{e}$ otherwise.
2. A unique valuation map, also denoted $v_{\mathcal{B}} : \mathcal{L} \rightarrow \mathcal{B}$, on the set of all ground formulas $\mathcal{L}$, according to the following conditions:
2.1. $v_{\mathcal{B}}(\sim X) = \sim v_{\mathcal{B}}(X)$
2.2. $v_{\mathcal{B}}(X \circledcirc Y) = v_{\mathcal{B}}(X) \circledcirc v_{\mathcal{B}}(Y)$, where $\circledcirc \in \{\wedge, \vee, \otimes, \oplus \leftarrow\}$
3. A truth assignment $u_B : \mathcal{L} \rightarrow \mathcal{B}$ will be called an extension of a truth assignment v_B if $u_B(\psi) \geq_k v_B(\psi)$ for all $\psi \in \mathcal{L}$. If u_B is an extension of v_B, we will write $u_B \geq_k v_B$.

The 'object' many-valued logic is based on four bilattice values which are *epistemic*. Sentences are to be marked with some of these bilattice logic values, according as to what the computer has been told; or, with only a slight metaphor, according to what it *believes or knows*. Of course these sentences *have* also Frege's ontological truth-values (true and false), independently of what the computer has been told: we want that the computer can use also these ontological 'meta' knowledge. Let, for example, the computer believes that the sentence p has a value $\top$ (possible); then the 'meta' sentence,"I (computer) believe that p has a possible value" is *ontologically true*. The many-valued encapsulation, defined as follows, is just the way to pass from the epistemic ('object') many-valued logic into ontological ('meta') 2-valued logic.
Such encapsulation is characterized by having capability for *semantic-reflection*: intuitively, for each predicate symbol we need some function which *reflects* its logic semantic over a domain Γ_U. Let introduce also the set of functional symbols κ_p over a domain Γ_U in our logical language in order to obtain an enriched logical language where we can encapsulate the 'object' (ordinary) many-valued logic programming. Such set of functional symbols will be derived from the following Bilattice-semantic mapping $\mathcal{K}$:

Definition 5. *A semantic-reflection is a mapping $\mathcal{K} : \mathcal{P}_S \rightarrow (\mathcal{B} \bigcup \{\boldsymbol{e}\})^{\bigcup_{i \leq \omega} \mathcal{T}_0^i}$, and we denote shortly $\kappa_p = \mathcal{K}(p) : \bigcup_{i \leq \omega} \mathcal{T}_0^i \rightarrow \mathcal{B} \bigcup \{\boldsymbol{e}\}$, $p \in \mathcal{P}_S$, such that for any $\boldsymbol{c} = (c_1, .., c_n) \in \mathcal{T}_0^n$, holds: $\kappa_p(\boldsymbol{c}) = \boldsymbol{e}$ iff $arity(p) \neq n$.*
If p is a built-in predicate, then a mapping κ_p is uniquely defined by: for any $\boldsymbol{c} \in \mathcal{T}_0^n$, $n = arity(p)$, holds that $\kappa_p(\boldsymbol{c}) = t$ if $p(\boldsymbol{c})$ is true; f otherwise.

5 Ontological encapsulation programming language

The many-valued ground atoms of a bilattice-based logical language $\mathcal{L}_{\mathcal{B}}$ can be transformed in 'encapsulated' atoms of a 2-valued logic in the following simple way: the original (many-valued) fact that the ground atom $A = p(c_1, .., c_n)$, of the n-ary predicate p, has an epistemic value $\alpha = \kappa_p(c_1, .., c_n)$ in $\mathcal{B}_4$, we transform in encapsulated atom $p^A(c_1, .., c_n, \alpha)$ with meaning "it is *true* that A has a value α". Indeed,

what we do is to *replace* the original n-ary predicate $p(x_1,..,x_n)$ with n+1-ary predicate $p^A(x_1,..,x_n,\alpha)$, with the added logic-attribute α. It is easy to verify that for any given many-valued valuation $v_{\mathcal{B}}$, every ground atom $p^A(c_1,..,c_n,\alpha)$ is ontologically true (when $\alpha = v_B(p(c_1,..,c_n))$) or false. Let EMV denote this new 2-valued encapsulation of many-valued logic *for logic programming*.

5.1 Syntax

We distinguish between what the reasoner believes in (at the *object* (epistemic many-valued sublanguage) level), and what is actually true or false in the real world (at the EMV ontological 'meta' level), thus, roughly, the 'meta' level is an (classic) encapsulation of the object level. Thus, we introduce the modal operator of encapsulation $\mathcal{E}$ as follows:

Definition 6. *Let P be an 'object' many-valued logic program with the set of predicate symbols P_S. The translation in the encapsulated syntax version in P^A is as follows:*
1. Each positive literal in P, $\mathcal{E}(p(x_1,..,x_n)) = p^A(x_1,..,x_n,\kappa_p(x_1,..,x_n))$;
2. Each negative literal in P, $\mathcal{E}(\sim p(x_1,..,x_n)) = p^A(x_1,..,x_n,\sim \kappa_p(x_1,..,x_n))$;
3. $\mathcal{E}(\phi \wedge \varphi) = \mathcal{E}(\phi) \wedge \mathcal{E}(\varphi)$;
4. $\mathcal{E}(\phi \vee \varphi) = \mathcal{E}(\phi) \vee \mathcal{E}(\varphi)$;
5. $\mathcal{E}(\phi \leftarrow \varphi) = \mathcal{E}(\phi) \leftarrow^A \mathcal{E}(\varphi)$ *, where $\leftarrow^A$ is a new syntax symbol for the implication at the encapsulated 2-valued 'meta' level.*
Thus, the obtained 'meta' program is equal to $P^A = \{\mathcal{E}(\phi) \mid \phi \text{ is a clause in } P\}$*, with the 2-valued Herbrand base* $H_P^A = \{ p^A(c_1,..,c_n,\alpha) \mid p(c_1,..,c_n) \in H_P \text{ and } \alpha \in \mathcal{B}\}$

This embedding of the many-valued 'object' logic program P into a 2-valued 'meta' logic program P^A is an *ontological* embedding: views formulae of P as beliefs and interprets negation $\sim p(x_1,..,x_n)$ in rather restricted sense - as belief in the falsehood of $p(x_1,..,x_n)$, rather as not believing that $p(x_1,..,x_n)$ is true (like in an ontological embedding for classical negation).
Like for Moore's autoepistemic operator, for the encapsulation modal operator $\mathcal{E}$, $\mathcal{E}\phi$ is intended to capture the notion of, "I know that ϕ has a value $v_B(\phi)$ ", for a given valuation v_B of the 'object' logic program.
Let $\mathcal{L}$ be the set of all ground well-formed formulae defined by this Herbrand base H_P and bilattice operations (included many-valued implication $\leftarrow$ also), with $\mathcal{B} \subseteq \mathcal{L}$. We define the set of all well-formed encapsulated formulae by:
$\mathcal{L}^A =_{def} \{\mathcal{E}(\psi) \mid \psi \in \mathcal{L}\}$, so that $H_P^A \subseteq \mathcal{L}^A$, thus, we can extend operator $\mathcal{E}$ to all formulas in $\mathcal{L}$ (also to bilattice logic values, such that $\mathcal{E} : \mathcal{B} \rightarrow 2$), so, we obtain

Proposition 3 *The encapsulation operator $\mathcal{E}$ is :*
1. Nondeductive modal operator, such that, for any $\alpha \in \mathcal{B}$, $\mathcal{E}(\alpha) = t$ if $\alpha = t$; f otherwise. It cannot be written in terms of the bilattice operations $\wedge, \vee, \otimes, \oplus$ and $\sim$.
2. Homomorphism between the 'object' algebra $(\mathcal{L}, \wedge, \vee, \leftarrow)$ with carrier set of (positive and negative) literals, and 'meta' algebra $(\mathcal{L}^A, \wedge^A, \vee^A, \leftarrow^A)$, where $\wedge^A, \vee^A$ are 2-valued reductions of bilattice meet and join, respectively, denoted by $\wedge, \vee$ also.

5.2 Semantics

The modal operator $\mathcal{E}$ is more selective than Moore's modal operator M (which returs the truth also when its argument has a possible value). In fact $M(\alpha) = \mathcal{E}(\alpha \vee \perp)$.
Notice, that with the transformation of the original 'object' logic program P into its annotated 'meta' version program P^A we obtain *always positive* consistent logic program.
A Herbrand interpretation of P^A is a 2-valued mapping $I^A : H_P^A \rightarrow \mathbf{2}$. We denote by $\mathbf{2}^{H_P^A}$ the set of all a-interpretations (functions) from H_P^A into $\mathbf{2}$, and by $\mathcal{B}^{H_P}$ the set of all *consistent* Herbrand many-valued interpretations, from H_P to the bilattice $\mathcal{B}$. The meaning of the *encapsulation* of this 'object' logic program P into this 'meta' logic program P^A is fixed into the kind of interpretation to give to such new introduced functional symbols $\kappa_p = \mathcal{K}(p)$: in fact *we want* [21] that they reflect (encapsulate) the semantics of the 'object' level logic program P.

Definition 7. *(Satisfaction) The* ***encapsulation*** *of an epistemic 'object' logic program P into an 'meta' program P^A means that, for any* consistent *many-valued Herbrand interpretation $I \in \mathcal{B}^{H_P}$ and its extension $v_B : \mathcal{L} \rightarrow \mathcal{B}$, the function symbols $\kappa_p = \mathcal{K}(p)$, $p \in P_S$ reflects this semantics (is compatible to it), i.e.*
for any tuple $\mathbf{c} \in \mathcal{T}_0^{arity(p)}$, $\kappa_p(\mathbf{c}) = I(p(\mathbf{c}))$.
So, we obtain a mapping, $\Theta : \mathcal{B}^{H_P} \rightarrow \mathbf{2}^{H_P^A}$*, such that* $I^A = \Theta(I) \in \mathbf{2}^{H_P^A}$ *with: for any ground atom* $p(\mathbf{c})$, $I^A(\mathcal{E}(p(\mathbf{c}))) = t$ *, if* $\kappa_p(\mathbf{c}) = I(p(\mathbf{c}))$; f *otherwise.*
Let g be a variable assignment which assigns values from Γ_U to object variables. We extent it to atoms with variables, so that $g(\mathcal{E}(p(x_1, .., x_n))) = \mathcal{E}(p(g(x_1), .., g(x_n)))$*, and to all formulas in the usual way:* ψ/g *denotes a ground formula obtained from* ψ *by assignment g, then*
1. $I^A \vDash_g \mathcal{E}(p(x_1, .., x_n))$ *iff* $\kappa_p((g(x_1), .., g(x_n))) = I(p(g(x_1), .., g(x_n)))$.
$I^A \vDash_g \mathcal{E}(\sim p(x_1, .., x_n))$ *iff* $\sim \kappa_p((g(x_1), .., g(x_n))) = I(p(g(x_1), .., g(x_n)))$.
2. $I^A \vDash_g \mathcal{E}(\phi \wedge \psi)$ *iff* $I^A \vDash_g \mathcal{E}(\phi)$ *and* $I^A \vDash_g \mathcal{E}(\psi)$.
3. $I^A \vDash_g \mathcal{E}(\phi \vee \psi)$ *iff* $I^A \vDash_g \mathcal{E}(\phi)$ *or* $I^A \vDash_g \mathcal{E}(\psi)$.
4. $I^A \vDash_g \mathcal{E}(\phi \leftarrow \psi)$ *iff* $v_B(\phi/g \leftarrow \psi/g)$ *is true.*

Notice that in this semantics the 'meta' implication $\leftarrow^A$, in $\mathcal{E}(\phi) \leftarrow^A \mathcal{E}(\psi) = \mathcal{E}(\phi \leftarrow \psi)$, is based on the 'object' epistemic many-valued implication $\leftarrow$ (which *is not classical*, i.e., $\phi \leftarrow \psi \neq \phi \vee \sim \psi$) and determines how the logical value of a body of clause "propagates" to its head.

Theorem 1 *The semantics of encapsulation $\mathcal{E}$ is obtained by identifying the semantic-reflection with the λ-abstraction of Generalized Herbrand interpretation, $\mathcal{K} = \lambda\mathcal{I}$, so that the semantics of many-valued logic programs can be determined by $\mathcal{I}$ (at 'object' level) or, equivalently, by its reflection $\mathcal{K}$ (at encapsulated or 'meta' level).*

Proof. From $\mathcal{K} = \lambda\mathcal{I}$ we obtain that for any $p(\mathbf{c}) \in H_P$ holds $I(p(\mathbf{c})) = \mathcal{I}(p, \mathbf{c}) = \lambda\mathcal{I}(p)(\mathbf{c}) = \mathcal{K}(p)(\mathbf{c}) = \kappa_p(\mathbf{c})$, what is the semantic of encapsulation.

We can consider the λ-abstraction of Generalized Herbrand interpretation as an epistemic semantics, because, given a Herbrand (epistemic) interpretation $I : H_P \rightarrow \mathcal{B}$,

then for any predicate symbol p and constant $\mathbf{c} \in \mathcal{T}_0^{arity(p)}$, holds $\lambda\mathcal{I}(p)(\mathbf{c}) = I(p(\mathbf{c}))$. Then the semantic of encapsulation may be defined as follows:

" ontological semantic-reflection $\equiv$ epistemic semantics", that is, $\mathcal{K} = \lambda\mathcal{I}$.

Recently, in [22], this semantics is used to give a coalgebraic semantics for logic programs. Notice that at 'meta' (ontological) level (differently from $\wedge, \vee$, which are classic 2-value boolean operators), the semantics for 'meta' implication operator, $I^A \vDash_g \mathcal{E}(\phi \leftarrow \psi)$, is not defined on $I^A \vDash_g \mathcal{E}(\phi)$ and $I^A \vDash_g \mathcal{E}(\psi)$. For example, let $I^A \vDash_g \mathcal{E}(p(\mathbf{c}))$ and $I^A \vDash_g \mathcal{E}(q(\mathbf{d}))$, with $\kappa_p(\mathbf{c}) = f$ and $\kappa_q(\mathbf{d}) = t$: then $p(\mathbf{c}) \leftarrow q(\mathbf{d})$ is false and, consequently, does not hold $I^A \vDash \mathcal{E}(p(\mathbf{c}) \leftarrow q(\mathbf{d}))$.

Proposition 4 *$I^A \vDash_g \mathcal{E}(\phi) \leftarrow^A \mathcal{E}(\psi)$ implies $I^A \vDash_g \mathcal{E}(\phi)$ and $I^A \vDash_g \mathcal{E}(\psi)$, but not viceversa. The truth of $\mathcal{E}(\phi/g)$ and $\mathcal{E}(\psi/g)$ are necessary but not sufficient conditions for the truth of $\mathcal{E}(\phi/g) \leftarrow^A \mathcal{E}(\psi/g)$.*

More over $\leftarrow^A$ has a *constructivistic* viewpoint (notice that the implication $\leftarrow^A$ is satisfied when the body and the head of such clause are *true*, while in the 'object' logic program such clause may be satisfied when their body and the head *are not true* also). Thus, by encapsulation of a many-valued 'object' logic program into a 2-valued 'meta' logic program we obtain a constructive logic program: in each clause we derive from the true facts in its body other new true facts.
Following the standard definitions, we say that an interpretation I^A, of a program P^A, is a *model* of a P^A if and only if every clause of P^A is satisfied in I^A. In this way we define a *model theoretic* semantics for encapsulated logic programs.
A set of formulas S, of encapsulated logic EMV, *logically entails* a formula ϕ, denoted $S \vDash \phi$, if and only if every model of S is also a model of ϕ.

6 Conclusion

We have presented a programming logic capable of handling inconsistent beliefs and based on the 4-valued Belnap's bilattice, which has clear model theory. In the process of the encapsulation we distinguish two levels: the 'object' many-valued level of ordinary logic programs with epistemic negation based on a bilattice operators, and the encapsulated or 'meta' logic programs. In this approach, 'inconsistent' logic program (which minimal stable models contain at least an 'inconsistent' ground atom) at object level is classic consistent logic program at 'meta' level also. In such abstraction we obtained a kind of a minimal Constructivistic Logic where fixpoint 'immediate consequence' operator is always continuous, and which is *computationally equivalent* to the standard Fitting's fixpoint semantics. Following this approach we are able to define a unique many-valued Herbrand model for databases with inconsistencies based on the fixpoint of a monotonic (w.r.t. knowledge ordering) immediate consequence operator, and the inference closure for many-valued logic programming also.
This research is partially supported by the project NoE INTEROP-IST-508011 and the project SEWASIE-IST-2001-3425. The autor wishes to thank Tiziana Catarci and Maurizio Lenzerini for their support.

References

1. M.Gelfond and V.Lifshitz, "The stable model semantics for logic programming," *In Proc. of the Fifth Logic Programming Symposium, Cambridge, MA. MIT Press*, pp. 1070–1080, 1988.
2. K.Fine, "The justification of negation as failure," *In Logic, Methodology and Philosophy of Science VIII, Amsterdam, North-Holland*, pp. 263–301, 1989.
3. M.C.Fitting, "A kripke/kleene semantics for logic programs," *Journal of Logic Programming 2*, pp. 295–312, 1985.
4. K.Kunen, "Negation in logic programming," *Journal of Logic Programming 4*, pp. 289–308, 1987.
5. T.Przymusinski, "Every logic program has a natural stratification and an iterated fixed point model," *In Eighth ACM Symposium on Principles of Databases Systems*, pp. 11–21, 1989.
6. T.Przymusinski, "Well-founded semantics coincides with three-valued stable-semantics," *Fundamenta Informaticae 13*, pp. 445–463, 1990.
7. G.Escalada Imaz and F.Manyá, "The satisfiability problem for multiple-valued horn formulae," *In Proc. International Symposium on Multiple-Valued Logics (ISMVL), Boston, IEEE Press, Los Alamitos*, pp. 250–256, 1994.
8. B.Beckert, R.Hanhle, and F.Manyá, "Transformations between signed and classical clause logic," *In Proc. 29th Int.Symposium on Multiple-Valued Logics, Freiburg,Germany*, pp. 248–255, 1999.
9. M.Kifer and E.L.Lozinskii, "A logic for reasoning with inconsistency," *Journal of Automated reasoning 9(2)*, pp. 179–215, 1992.
10. M.Kifer and V.S.Subrahmanian, "Theory of generalized annotated logic programming and its applications," *Journal of Logic Programming 12(4)*, pp. 335–368, 1992.
11. M.Ginsberg, "Multivalued logics: A uniform approach to reasoning in artificial intelligence," *Computational Intelligence, vol.4*, pp. 265–316, 1988.
12. M.C.Fitting, "Billatices and the semantics of logic programming," *Journal of Logic Programming,11*, pp. 91–116, 1991.
13. M.H.van Emden, "Quantitative deduction and its fixpoint theory," *Journal of Logic Programming,4,1*, pp. 37–53, 1986.
14. S.Morishita, "A unified approach to semantics of multi-valued logic programs," *Tech. Report RT 5006, IBM Tokyo*, 1990.
15. V.S.Laksmanan and N.Shiri, "A parametric approach to deductive databases with uncertainty," *IEEE Transactions on Knowledge and Data Engineering,13(4)*, pp. 554–570, 2001.
16. A.Calì, D.Calvanese, G.De Giacomo, and M.Lenzerini, "Data integration under integrity constraints," in *Proc. of the 14th Conf. on Advanced Information Systems Engineering (CAiSE 2002)*, 2002, pp. 262–279.
17. Z. Majkić, "Fixpoint semantic for query answering in data integration systems," *AGP03 - 8.th Joint Conference on Declarative Programming, Reggio Calabria*, pp. 135–146, 2003.
18. N.D.Belnap, "A useful four-valued logic," *In J-M.Dunn and G.Epstein, editors, Modern Uses of Multiple-Valued Logic. D.Reidel*, 1977.
19. M.C.Fitting, "Billatices are nice things," *Proceedings of Conference on Self-Reference, Copenhagen*, 2002.
20. M.C.Fitting, "Kleene's three valued logics and their children," *Fundamenta Informaticae*, vol. 26, pp. 113–131, 1994.
21. Z. Majkić, "Two-valued encapsulation of many-valued logic programming," *Technical Report, University 'La Sapienza', Roma, in http://www.dis.uniroma1.it/ $\sim$ majkic/.*, 2003.
22. Z. Majkić, "Coalgebraic semantics for logic programming," *18th Worshop on (Constraint) Logic Programming, WLP 2004, March 04-06, Berlin, Germany*, 2004.

Frequent Pattern Queries for Flexible Knowledge Discovery

Francesco Bonchi[1], Fosca Giannotti[1], and Dino Pedreschi[2]

[1] ISTI - CNR, Area della Ricerca di Pisa, Via Giuseppe Moruzzi, 1 - 56124 Pisa, Italy
[2] Dipartimento di Informatica, Via F. Buonarroti 2, 56127 Pisa, Italy

Abstract. In this paper we study data mining query language and optimizations in the context of a Logic-based Knowledge Discovery Support Environment. i.e., a flexible knowledge discovery system with capabilities to obtain, maintain, represent, and utilize both induced and deduced knowledge. In particular, we focus on frequent pattern queries, since this kind of query is at the basis of many mining tasks, and it seems appropriate to be encapsulated in a knowledge discovery system as a primitive operation. We introduce an inductive language for frequent pattern queries, which is simple enough to be highly optimized and expressive enough to cover the most of interesting queries. Then we define an optimized constraint-pushing operational semantics for our inductive language. This semantics is based on a frequent pattern mining operator, which is able to exploit as much as possible the given set of constraints, and which can adapt its behavior to the characteristics of the given input set of data.

1 Introduction

Knowledge Discovery in Databases is a complex iterative and interactive process which involves many different tasks that can bring the analyst from raw dirty data to actionable knowledge. A rigorous user interaction during such process is needed in order to facilitate efficient and fruitful knowledge manipulation and discovery. Such a rigorous interaction can be achieved by means of a set of *data mining primitives*, that should include the specification of the source data, the kind of knowledge to be mined, background or domain knowledge, interestingness measures for patterns evaluation, and finally the representation of the extracted knowledge. Providing a query language capable to incorporate all these features may result, like in the case of relational databases, in a high degree of expressiveness in the specification of data mining tasks, a clear and well-defined separation of concerns between logical specification and physical implementation of data mining tasks, and easy integration with heterogeneous information sources.

Clearly the implementation of this vision presents a great challenge. A path to this goal is indicated in [10] where Mannila introduces an elegant formalization for the notion of interactive mining process: the term *inductive database* refers to a relational database plus the set of all sentences from a specified class of sentences that are true of the data.

Definition 1. Given an instance $\mathbf{r}$ of a relation $\mathbf{R}$, a class $\mathcal{L}$ of sentences (patterns), and a selection predicate q, a pattern discovery task is to find a theory

$$\mathcal{T}h(\mathcal{L}, \mathbf{r}, q) = \{s \in \mathcal{L} | q(\mathbf{r}, s) \text{ is true}\}$$

The selection predicate q indicates whether a pattern s is considered interesting, and it is defined as a conjunction of *constraints* defined by the analyst. In other words,

the inductive database is a database framework which integrates the raw data with the knowledge extracted from the data and materialized in the form of patterns. In this way, the knowledge discovery process consists essentially in an iterative querying process, enabled by a query language that can deal either with raw data or patterns.

1.1 Logic-based Knowledge Discovery Support Environment

The notion of inductive database fits naturally in rule-based languages, such as *deductive databases* [6]. A deductive database can easily represent both extensional and intensional data, thus allowing a higher degree of expressiveness than traditional relational algebra. Such capability makes it viable for suitable representation of domain knowledge and support of the various steps of the knowledge discovery process.

The last consideration leads to the definition of a *Logic-based Knowledge Discovery Support Environment* (LKDSE in the following) as a deductive database programming language equipped with inductive rules. The main idea of the previous definition is that of providing a simple way for modelling the key aspects of a data mining query language:

- the source data and the background knowledge are represented by the relational extensions;
- deductive rules provide a way of integrating background knowledge in the discovery process, pre-processing source data, post-processing and reasoning on the newly extracted knowledge;
- inductive rules provide a way of declaratively invoking mining procedures with explicit representation of interestingness measures.

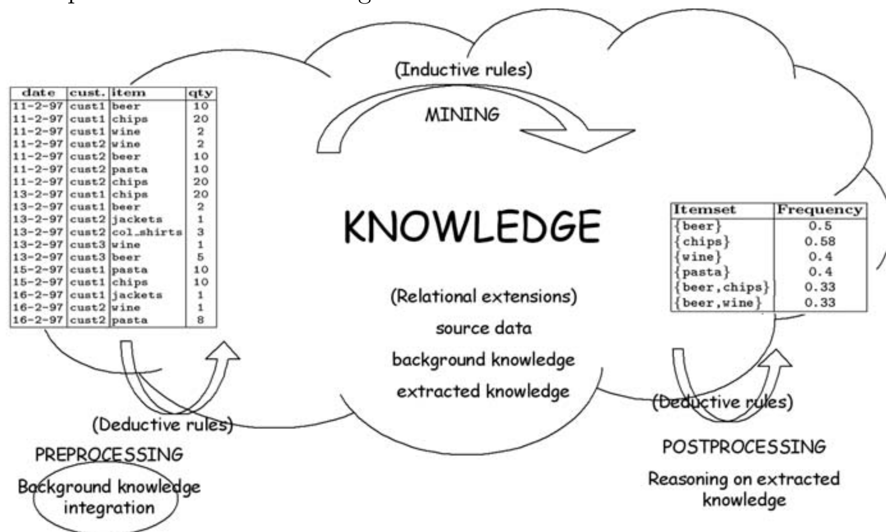


Fig. 1. The vision of Logic-based Knowledge Discovery Support Environment.

The main problem of a deductive approach is how to choose a suitable representation formalism of the inductive part, capable of expressing the correspondence between the deductive part and the inductive part. More specifically, the problem is how to formalize the specification of the set $\mathcal{L}$ of patterns in a way such that each pattern $s \in \mathcal{T}h(\mathcal{L}, \mathbf{r}, q)$ is represented as an independent (logical) entity (i.e., a predicate) and each manipulation of $\mathbf{r}$ results in a corresponding change in s.

A first attempt to define inductive rules on a deductive database is in [6]. In this work the notion of inductive rules in a deductive framework is elegantly defined by means of *user-defined aggregates* on the Datalog++ logic-based database language and its practical implementation, namely the $\mathcal{LDL}$++ deductive database system. The resulting LKDSE has been named $\mathcal{LDL}$-$\mathcal{M}$ine. For lack of space, we shall omit a presentation of $\mathcal{LDL}$++, and confine ourselves to mention that it is a rule-based language with a Datalog-like syntax, and a semantics that extends that of relational database query languages with recursion [7].

The main drawback of using user defined aggregates as a mean to define inductive queries is the atomicity of the aggregate that makes us loose optimization opportunities. For instance, in frequent pattern discovery a user may wish to mine only frequent patterns or rules that satisfy some constraints. This constraints could be used to reduce the search space of the computation as shown in many works in literature [11, 9, 5, 3, 4]. Unluckily in this approach such constraints can not be directly exploited by the inductive rules but can only be checked by deductive rules after the extraction of the frequent itemsets. In order to exploit constraints to reduce the search space of the mining algorithm one should redefine the mining aggregate for any particular constraint and query. This requires nontrivial programming effort for the analyst and however, constraints satisfaction would be behind the query level, thus losing the transparency which is one main requirement of inductive database.

1.2 Our Position and Objective

The objective of this research is to study *pattern discovery queries* (or *mining queries* or *inductive queries*), both by the point of view of the language and of the optimizations [2], in the context of a Logic-based Knowledge Discovery Support Environment based on Datalog++ as deductive database language, equipped with inductive rules.

In our vision, the data analyst should have a high-level vision of the data mining system, without worrying about the details of the computational engine, in the very same way a database designer has not to worry about query optimizations. The analyst must be provided with a set of primitives to be used to communicate with the data mining system, using a data mining query language. The analyst just needs to declaratively specify in the data mining query how the desired patterns should look like and which conditions they should obey (a set of constraints). Indeed, the task of composing all constraints and producing the most efficient mining strategy (execution plan) for the given data mining query should be left to an underlying *query optimizer*. This is the paradigm of *Declarative Mining*. Following this vision, we define a new language of inductive queries on top of a deductive database. In our framework an inductive rule is simply a conjunction of sentences about the desired patterns:

- These sentences are taken form a specified class of sentences and they can be seen as mining primitives, computed by a specialized algorithm (and not by aggregates as in the previous approach).
- The set of all admissible sentences is just some ”syntactic sugar” on top of an algorithm. The algorithm is the *inductive engine*.
- Each sentence can be defined over some deductive predicates (relations) defining the data source for the mining primitive.

Therefore, in this setting we have a clear distinction between what must be computed by deductive engine (deductive rules) and what by the inductive engine (inductive rules). Moreover we clearly specify the relationship between the inductive and the deductive part of an inductive database: the deductive engine feeds the inductive engine with data sources by means of the deductive predicates contained in the inductive sentences; the inductive engine returns in the deductive database predicates representing patterns that satisfy all the sentences in the inductive rule. Having a well defined and restricted set of possible sentences allows us to write highly optimized algorithms to compute inductive rules.

In this paper, we focus on *frequent pattern queries* over a transactional database [1, 8], i.e. queries which model the pattern discovery task $\mathcal{T}h(\mathcal{L}, \mathbf{r}, q)$, where $\mathcal{L}$ is the pattern domain of itemsets, and q is a minimal frequency constraint, or in other words, q selects itemsets which which appear in the transactional database a number of times greater than a user-defined minimum frequency threshold.

The rationale behind this choice is that this kind of query is a simple primitive operation (nothing more than counting) which is at the basis of practically every mining tasks, but which is usually the most time-consuming operation in any mining session. Therefore, it seems appropriate to encapsulate frequency counting in a knowledge discovery system as a primitive operation, and to study its optimizations.

2 Frequent Pattern Mining

Definition 2 (Frequent Pattern Mining). Let $\mathcal{I} = \{x_1, ..., x_n\}$ be a set of distinct literals, usually called *items*, where an item is an object with some predefined attributes (e.g., price, type, etc.). An *itemset* X is a non-empty subset of $\mathcal{I}$. If $|X| = k$ then X is called a *k-itemset*. A constraint on itemsets is a function $\mathcal{C} : 2^{\mathcal{I}} \rightarrow \{true, false\}$. We say that an itemset I satisfies a constraint if and only if $\mathcal{C}(I) = true$. We define the *theory* of a constraint as the set of itemsets which satisfy the constraint: $Th(\mathcal{C}) = \{X \in 2^{\mathcal{I}} \mid \mathcal{C}(X)\}$. A *transaction database* $\mathcal{D}$ is a bag of itemsets $t \in 2^{\mathcal{I}}$, usually called *transactions*. The *cover* of an itemset X in database $\mathcal{D}$, is the set of transactions in $\mathcal{D}$ which are superset of X: $cov_{\mathcal{D}}(X) = \{t \in \mathcal{D} \mid t \supseteq X\}$. The *support* of an itemset X in database $\mathcal{D}$, denoted $supp_{\mathcal{D}}(X)$, is the cardinality of $cov_{\mathcal{D}}(X)$. Given a user-defined *minimum support* σ, an itemset X is called *frequent* in $\mathcal{D}$ if $supp_{\mathcal{D}}(X) \geq \sigma$. This defines the minimum frequency constraint: $\mathcal{C}_{freq[\mathcal{D},\sigma]}(X) \Leftrightarrow supp_{\mathcal{D}}(X) \geq \sigma$. When the dataset and the minimum support threshold are clear from the context, we indicate the frequency constraint simply $\mathcal{C}_{freq}$. Thus with this notation, the *frequent itemsets mining problem* requires to compute the set of all frequent itemsets $Th(\mathcal{C}_{freq})$. In general given a conjunction of constraints $\mathcal{C}$ the *constrained frequent itemsets mining problem* requires to compute $Th(\mathcal{C}_{freq}) \cap Th(\mathcal{C})$.

Example 3 (Market Basket Analysis). The most natural way to think about a transaction database is the *sales database* of a retail store, where the content of each basket appearing at the cash register is recorded. In this context a transaction $\langle tid, X\rangle$ represents a basket identifier and its content. A transaction database can be represented also as a relational table as shown in Figure 2(a). In that table a transaction or basket identifier is not explicitly given, but for instance, one could use the couple $(\texttt{date}, \texttt{cust})$

date	cust.	item	qty
11-2-97	cust1	beer	10
11-2-97	cust1	chips	20
11-2-97	cust1	wine	2
11-2-97	cust2	wine	2
11-2-97	cust2	beer	10
11-2-97	cust2	pasta	10
11-2-97	cust2	chips	20
13-2-97	cust1	chips	20
13-2-97	cust1	beer	2
13-2-97	cust2	jackets	1
13-2-97	cust2	col_shirts	3
13-2-97	cust3	wine	1
13-2-97	cust3	beer	5
15-2-97	cust1	pasta	10
15-2-97	cust1	chips	10
16-2-97	cust1	jackets	1
16-2-97	cust2	wine	1
16-2-97	cust2	pasta	8
16-2-97	cust3	chips	20
16-2-97	cust3	col_shirts	3
16-2-97	cust3	brown_shirts	2
18-2-97	cust1	pasta	5
18-2-97	cust1	wine	1
18-2-97	cust1	chips	20
18-2-97	cust1	beer	10
18-2-97	cust2	beer	12
18-2-97	cust2	beer	10
18-2-97	cust2	chips	20
18-2-97	cust2	chips	20
18-2-97	cust3	pasta	10

(a)

date	cust.	itemset
11-2-97	cust1	{beer,chips,wine}
11-2-97	cust2	{wine,beer,pasta,chips}
13-2-97	cust1	{chips,beer}
13-2-97	cust2	{jackets,col_shirts}
13-2-97	cust3	{wine,beer}
15-2-97	cust1	{pasta,chips}
16-2-97	cust1	{jackets}
16-2-97	cust2	{wine,pasta}
16-2-97	cust3	{chips,col_shirts,brown_shirts}
18-2-97	cust1	{pasta,wine,chips,beer}
18-2-97	cust2	{beer,chips}
18-2-97	cust3	{pasta}

(b)

item	price	type
beer	10	beverage
chips	3	snack
wine	20	beverage
pasta	2	food
jackets	100	clothes
col_shirt	30	clothes
brown_shirt	25	clothes

(c)

Fig. 2. (a) A sample `sales` table and (b) one of its transactional representations; (c) the `product` table.

to indicate it. If our relational language allows set-valued fields, we can have the same relation in its transactional representation as in Figure 2(b).

In classical frequent pattern mining, the popular Apriori algorithm [1] exploits an interesting property of frequency for pruning the exponential search space of the problem: whenever the support of an itemset violates the frequency constraint, then all its supersets can be pruned away from the search space, since they will violate the frequency constraint too. This property of frequency is called *antimonotonicity* (see Definition 4) and is the basis of the breadth-first level-wise (from small itemsets to large itemsets) Apriori exploration and pruning of the search space.

Definition 4 (Antimonotone constraint). Given an itemset X, a constraint $\mathcal{C}_{AM}$ is *antimonotone* if $\forall Y \subseteq X : \mathcal{C}_{AM}(X) \Rightarrow \mathcal{C}_{AM}(Y)$.

As already stated frequency is clearly an antimonotone constraint. Many other kind of constraints with the same nice property can be defined. For instance one could be interested in mining frequent itemsets with a total sum of prices $\leq 50\$$: this constraint is antimonotone because any itemset that already has a sum of prices greater than 50$ will never produce a solution. Adding more items to the itemset will simply make it more expensive, so it will never satisfy the constraint. Such constraints can be pushed deeply down into the frequent pattern mining computation since they behave exactly as the frequency constraint: if they are not satisfiable at an early level (small patterns), they have no hope of becoming satisfiable later (larger patterns). Moreover, since any

conjunction of antimonotone constraints is antimonotone as well, they can be exploited all together, in the same way of frequency, to prune the search space. The more antimonotone constraints the user specifies, the more selective the search will be.

The case is more subtle for constraints which exhibit the opposite property to antimonotonicity.

Definition 5 (Monotone constraint). Given an itemset X, a constraint $\mathcal{C}_M$ is *monotone* if: $\forall Y \supseteq X : \mathcal{C}_M(X) \Rightarrow \mathcal{C}_M(Y)$.

Since the frequency computation moves from small to large patterns, we can not push monotone constraints directly in it. At an early stage, if an itemset is too small or too cheap to satisfy a monotone constraint, we can not yet say nothing about its supersets. Perhaps, just adding a very expensive single item to the itemsets could raise the total sum of prices over the given threshold, thus making the resulting itemset satisfy the monotone constraint. For this reason the monotone constraints have always been considered *"hard to push"*, until the recent proposal of ExAnte [5]. In that work we have shown how monotone constraints can be exploited together with the frequency constraints by means of data-reduction. A transaction which does not satisfy a monotone constraint (recall here that a transaction is an itemset) can be deleted by the transaction database. This way, pushing monotone constraints does not reduce antimonotone pruning opportunities, on the contrary, such opportunities are boosted. Dually, pushing antimonotone constraints boosts monotone pruning opportunities: the two components strengthen each other recursively. This idea has been generalized in an Apriori-like computation in ExAMiner [4].

A succinct constraint $\mathcal{C}_S$ is such that, whether an itemset X satisfies it or not, can be determined based on the singleton items which are in X. Informally, given A_1, the set of singleton items satisfying a succinct constraint $\mathcal{C}_S$, then any set X satisfying $\mathcal{C}_S$ is based on A_1 , i.e. X contains a subset belonging to A_1 (for the formal definition of succinct constraints see [11]). A $\mathcal{C}_S$ constraint is *pre-counting pushable*, i.e. it can be satisfied at candidate-generation time: these constraints are pushed in the level-wise computation by substituting the usual *generate_apriori* procedure, with the proper (w.r.t. $\mathcal{C}_S$) candidate generation procedure.

In Section 5 we describe how these properties of constraints are exploited by the optimized operational semantics of our framework.

3 Frequent Pattern Queries

In this Section going through a rigorous identification of all its basic components we provide a definition of frequent pattern query, i.e. a query defining a frequent pattern mining task over a relational database $\mathcal{D}$.

Definition 6 (Mining View). Given a database $\mathcal{D}$ a relation $\mathcal{V}$ derived from $preds(\mathcal{D})$, explicitly indicated in the frequent pattern query as data source, is named *mining view.*

Definition 7 (Transaction id). Given a database $\mathcal{D}$ and a relation $\mathcal{V}$ derived from $preds(\mathcal{D})$. Let $\mathcal{V}$ with attributes $sch(\mathcal{V})$ be our mining view. Any subset of attributes $\mathcal{T} \subset sch(\mathcal{V})$ can be used as *transaction id.*

Definition 8 (Circumstance attribute). Given a database $\mathcal{D}$ and a relation $\mathcal{V}$ derived from $preds(\mathcal{D})$. Let $\mathcal{V}$ with attributes $sch(\mathcal{V})$ be our mining view. Given a subset of attributes $\mathcal{T} \subset sch(\mathcal{V})$ as transaction id, we define any attribute $A \in sch(R)$ where R is a relation in $preds(D)$ *circumstance attribute* provided that $A \notin \mathcal{T}$ and the functional dependency $\mathcal{T} \to A$ holds for in $\mathcal{D}$.

Definition 9 (Item attribute). Given a database $\mathcal{D}$ and a relation $\mathcal{V}$ derived from $preds(\mathcal{D})$. Let $\mathcal{V}$ with attributes $sch(\mathcal{V})$ be our mining view. Given a subset of attributes $\mathcal{T} \subset sch(\mathcal{V})$ as transaction id, let $Y = \{y | y \in sch(\mathcal{V}) \backslash \mathcal{T} \wedge \mathcal{T} \to y \textit{ does not hold}\}$; we define an attribute $\mathcal{A} \in Y$ an *item attribute* provided the functional dependency $\mathcal{T}\mathcal{A} \to Y \backslash \mathcal{A}$ holds in $\mathcal{D}$.

Proposition 10. *Given a relational database $\mathcal{D}$, a triple $\langle \mathcal{V}, \mathcal{T}, \mathcal{I} \rangle$ denoting the mining view $\mathcal{V}$, the transaction id $\mathcal{T}$, the item attribute $\mathcal{I}$, uniquely identifies a transactional database, as defined in Definition 2.*

Definition 11 (Descriptive attribute). Given a database $\mathcal{D}$ and a relation $\mathcal{V}$ derived from $preds(\mathcal{D})$. Let $\mathcal{V}$ with attributes $sch(\mathcal{V})$ be our mining view. Given a subset of attributes $\mathcal{T} \subset sch(\mathcal{V})$ as transaction id, and given $\mathcal{A}$ as item attribute; we define *descriptive attribute* any attribute $X \in sch(R)$ where R is a relation in $preds(\mathcal{D})$, provided the functional dependency $\mathcal{T}\mathcal{A} \to X$ holds in $\mathcal{D}$.

Consider the mining view: `sales(tID, locationID, time, product, quantity)` where each attribute has the intended semantics of its name and with `tID` acting as the transaction id. Since the functional dependency {`tID`} $\to$ {`locationID`} holds, `locationID` is a circumstance attribute. The same is true for `time`. We also have {`tID, product`} $\to$ {`quantity`}, thus `product` is an item attribute, while `quantity` is a descriptive attribute.

Note that, from the previous definitions, *transaction id* and the *item attribute* must be part of the mining view, while circumstance and descriptive attributes could be also in other relations. Constraints, as introduced in the previous section, are defined on item attributes and descriptive attributes. Constraints over the *transaction id* or over circumstance attributes are not real constraints since they can be seen as selection conditions on the transactions to be mined and thus they can be satisfied in the definition of the *mining view*. Next definition lists all kinds of constraint that we admit in our framework.

Constraint	Description
$s \supseteq \{a_1, \ldots, a_n\}$	itemset contains
$s \subseteq \{a_1, \ldots, a_n\}$	itemset domain
$\lvert s \rvert \, \theta \, m$	cardinality constraint
$s.d \supseteq S$	descriptive attribute contains
$s.d \subseteq S$	descriptive attribute domain
$aggr\{d \mid i \in s \wedge P(i, \ldots, d)\} \, \theta \, m$	aggregate on descriptive attribute

Fig. 3. Bound constraints for frequent pattern query.

Definition 12 (Bound constraints). Given a database $\mathcal{D}$, a mining view $V(t, i, \ldots)$ where t indicates the *transaction id* and i denotes the *item attribute*, a relation $P(i, \ldots, d) \in preds(\mathcal{D})$ where d indicates a descriptive attribute, all kinds of constraint admitted in a frequent pattern query are listed in Figure 3. The following notation is adopted:

- s is an itemset;
- $a_1, \ldots, a_n$ are items;
- $d_1, \ldots, d_n$ are numeric constants of a descriptive attribute;
- m is a numeric constant;
- S is a set-valued or discrete constant;
- $\theta \in \{\leq, \geq, =\}$
- $aggr \in \{min, max, sum, avg, count, range\}$

Definition 13 (Frequent pattern query). Given a database $\mathcal{D}$, a frequent pattern query is a quintuple $\langle \mathcal{V}, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C} \rangle$ denoting the mining view $\mathcal{V}$, the transaction id $\mathcal{T}$, the item attribute $\mathcal{I}$, the minimum support threshold σ, and a conjunction of bound constraints $\mathcal{C}$.

The result of a frequent pattern query is a binary relation recording the set of itemset which satisfy $\mathcal{C}$ and are frequent in the transaction database $\langle \mathcal{V}, \mathcal{T}, \mathcal{I} \rangle$ w.r.t. σ and their supports:

$$freq_{\langle \mathcal{V}, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C} \rangle}(I, S) \equiv \{(I, S) \mid \mathcal{C}(I) \ \wedge \ supp_{\langle \mathcal{V}, \mathcal{T}, \mathcal{I} \rangle}(I) = S \ \wedge \ S \geq \sigma\}$$

Example 14. A frequent pattern query for the `sales` table in Figure 2 (a), and the `product` table in Figure 2 (c), querying itemsets having a support ≥ 3 (transactions are made grouping by customer and date), and having a total price ≥ 30, could be simply defined as:

$$freq_{\langle sales, \{date, cust\}, item, 3, sum\{p \mid i \in I \wedge product(i,p,t)\} \geq 30 \rangle}(I, S) \equiv$$

$$\{(I, S) \mid sum\{p \mid i \in I \wedge product(i, p, t)\} \geq 30 \wedge supp_{\langle sales, \{date, cust\}, item \rangle}(I) = S \wedge S \geq 3\}$$

The result of such query is a relation `(I,S)` with the following two entries: `({beer, wine},4)` and `({beer, wine, chips},3)`.

4 Inductive Rules for Frequent Patterns

In this Section we provide the syntactic sugar to express the frequent pattern queries defined above. As already stated, in our language we drop the *user-defined aggregates* approach of $\mathcal{LDL}$-$\mathcal{M}$ine, and we define an inductive rule simply as a conjunction of sentences about the desired patterns (a conjunction of constraints).

In the following we provide a brief overview of the terms of our language to express frequent pattern queries. For the full syntax of our language, as well as for the definition of safe rules, see [2].

Definition 15. An inductive rule is a rule $H \leftarrow B_1, \ldots B_n$ such that:

- $B_1, \ldots B_n$ is a conjunction of inductive sentences, possibly containing deductive predicates.

– H is the predicate representing the induced pattern.

Since we are modelling frequent pattern queries, a frequency inductive sentence will always be present in the body of an inductive rule.

Definition 16 (Frequency inductive sentence). Let P be a variable for the induced frequent pattern, and TDB a transaction database (a set of sets), then $freq(P, TDB)$ is the frequency inductive sentence which computes $supp_{TDB}(P)$.

Since we are dealing with frequent itemsets, sets will be first class citizens on which inductive sentences will be defined. For the same reason, traditional set operations ($\subseteq, \in, \notin$) will be useful to write inductive rules. A set can be a set of numbers, a set of strings, a set of variables or a set of sets. Moreover, since we want to compute patterns from transaction databases which can be in relational form, a set of sets can be built by grouping values of an attribute, as in the following Definition.

Definition 17 (Pseudo-aggregation sentence). Given a relation R such that $X \in sch(R)$ and $Y \subset sch(R)$. The sentence $\langle X|Y \rangle$ denotes the pseudo aggregation on attribute X grouping by attributes in Y.

In other words, if the term $\langle \mathtt{I}|\{\mathtt{D}, \mathtt{C}\}\rangle$ appears in an inductive rule containing the predicate `sales(D,C,I,Q)`; it corresponds to the pseudo-aggregate defined by the Datalog++ deductive rule:

$$\mathtt{sales}(\mathtt{D}, \mathtt{C}, \langle \mathtt{I} \rangle) \leftarrow \mathtt{sales}(\mathtt{D}, \mathtt{C}, \mathtt{I}, \mathtt{Q}).$$

which transforms the table in Figure 2(a) in the table in Figure 2(b).

Finally, in our inductive query language we need to express aggregates on descriptive attributes, in order to write constraints as those ones introduced in the previous section. The following example clarifies the relationship between the inductive and the deductive part of an inductive database, as well as the difference between the aggregate-based approach and our framework.

Example 18. Consider the frequent pattern query in Example 14: find itemsets having a support ≥ 3 (transactions are made grouping by customer and date), and having a total price ≥ 30.

In our framework we have two choices:

1. write an inductive rule to compute all frequent itemsets having a support ≥ 3 and a deductive to selects those frequent itemsets which satisfy the constraint of having a total price ≥ 30;
2. write the constraint directly in the body of the inductive rule defining the interesting patterns.

By a semantical point of view the two approaches are equivalent: they give the same result. By a purely computational point of view the first choice corresponds to a *generate-and-test* approach, while the second choice corresponds to a *constraint pushing* approach which is much more efficient.

In the aggregate-based approach of $\mathcal{LDL}$-$\mathcal{M}$ineonly the first choice is possible. We need a pseudo-aggregate rule to de-normalize the data source, the inductive rule based on the user defined `patterns` aggregate, a rule based on the sum aggregate to compute total sum of prices, and finally a rule to select the resulting patterns.

$$
\begin{array}{ll}
\mathtt{sales(D,C,\langle I\rangle)} & \leftarrow \mathtt{sales(D,C,I,Q).} \\
\mathtt{frequentPatterns(patterns\langle(3,S)\rangle)} & \leftarrow \mathtt{sales(D,C,S).} \\
\mathtt{sumFP(S,N,sum\langle P\rangle)} & \leftarrow \mathtt{frequentPatterns(S,N),item(L,T,P),member(L,S).} \\
\mathtt{answer(S,N)} & \leftarrow \mathtt{sumFP(S,N,SP),SP >= 30.}
\end{array}
$$

In our language, we can express that query with the following inductive rule:

$$
\begin{array}{ll}
\mathtt{frequent_pattern(S,N)} \leftarrow & \mathtt{N = freq(S,X),\ X = \langle I|\{D,C\}\rangle,\ sales(D,C,I,Q),} \\
& \mathtt{N >= 3,\ L \in S,\ sum(P,product(L,P,T)) >= 30.}
\end{array}
$$

Observe how in this rule all the basic components of a frequent pattern query are expressed. First of all we have the *mining view* declaratively indicated as a predicate in the body of the rule. Then we have the *transaction id* expressed in the right part of the pseudo-aggregation term, and the *item attribute* in the left part. Moreover we have two variables to indicate computed patterns and their support, which is constrained to be no less than a given *min-sup* threshold.

One could wish to return as result of a frequent pattern query additional information regarding the induced pattern. Therefore we allow inductive rules with more than two variables in the head. For sake of consistency, the additional variables must be aggregated at the itemset level. In other words, only variables describing aggregation of descriptive attributes are allowed.

For instance, we can ask to return together with itemsets and their supports, also the total sum of prices of an itemset:

$$
\begin{array}{ll}
\mathtt{frequent_pattern(S,N,TP)} \leftarrow & \mathtt{N = freq(S,X),\ X = \langle I|\{D,C\}\rangle,\ sales(D,C,I,Q),} \\
& \mathtt{N >= 3,\ L \in S,\ TP = sum(P,product(L,P,T)),\ TP >= 30.}
\end{array}
$$

As suggested by Example 18, each inductive rule is equivalent to one or more Datalog++ deductive rules. This observation is used to provide a declarative formal semantics for our language. In [2] formal semantics of the new inductive rules is provided by showing that exists a unique mapping from each safe inductive rule of the language to a Datalog++ program with aggregates (we shall omit details for lack of space). Thanks to this mapping we can define the formal declarative semantics of an inductive rule as the *iterated stable model* of the corresponding Datalog++ program [7].

The next example shows how to define an association rules mining tasks in our framework. More complex examples of frequent pattern queries can be found in [2].

Example 19 (Association Rules). It is well known that the task of computing association rules is performed by dividing it in two subproblems. In the first one, which is the real mining task, itemsets which are frequent for the given minimum support threshold are computed. In the second subproblem, which can be seen as a post-processing filtering task, the available association rules, for the given confidence threshold, are extracted

from the frequent itemsets computed during the mining phase. In our framework we compute frequent itemsets by means of an inductive rule, and association rules by means of a deductive Datalog++ rule.

Suppose we want to compute simple association rules from the table in Figure 2(a), grouping transactions by day and customer, and having support more than 4 and confidence more than 0.6. The first rule is an inductive rule which defines the computation of patterns with an absolute support greater than 4.

$$\begin{aligned}
\texttt{frequentPatterns}(\texttt{Set},\texttt{Supp}) &\leftarrow \texttt{Supp} = \texttt{freq}(\texttt{Set},\texttt{X}),\ \texttt{sales}(\texttt{D},\texttt{C},\texttt{I},\texttt{Q}),\\
&\quad\ \texttt{X} = \langle \texttt{I} | \{\texttt{D},\texttt{C}\}\rangle,\ \texttt{Supp} >= 4.\\
\texttt{rules}(\texttt{Left},\texttt{Right},\texttt{Supp},\texttt{Conf}) &\leftarrow \texttt{frequentPatterns}(\texttt{A},\texttt{Supp}),\texttt{frequentPatterns}(\texttt{R},\texttt{S}_1),\\
&\quad\ \texttt{subset}(\texttt{R},\texttt{A}),\texttt{difference}(\texttt{A},\texttt{R},\texttt{L}),\texttt{Conf} = \texttt{Supp/S1},\texttt{Conf} >= 0.6.
\end{aligned}$$

The second rule is a Datalog++ deductive rule which computes the available association rules from the frequent itemsets (the result of the evaluation of the first rule is in Figure 4(a), while the result of the evaluation of the second rule is in Figure 4(b)).

5 Constraint Pushing Optimization

In Section 2 we have briefly reviewed constraints in frequent pattern mining and their properties. Such properties can be exploited for pruning the search space, and hence obtaining an optimized evaluation of frequent pattern queries. Following this approach, we systematically analyze each kind of constraint admitted in our language, and based on its properties, we define the constraint manager operator $\mathcal{CM}$ for the constraint pushing in the computation.

Similarly to the work done in [11], we divide all other kind of constraints in classes according to their properties. The main difference here is that, for us, monotone constraints are no longer hard constraints, since we have developed efficient techniques to push them in the computation, as those ones described in [3–5]. Therefore, we distinguish between 5 classes of constraints:

$\mathcal{C}_{AM}$ **constraints that are antimonotone but not succinct:** antimonotone constraints are exploited as usual to prune the search space in the level-wise, Apriori-like computation. They are checked together with frequency as a unique antimonotone constraint. Moreover, they are used in a data reduction fashion too, exploiting the real amalgam of antimonotonicity and monotonicity as described in [4, 5].

Set	Supp
{beer}	6
{chips}	7
{wine}	5
{pasta}	5
{beer,chips}	4
{beer,wine}	4

(a)

Left	Right	Supp	Conf
{beer}	{chips}	4	0.66
{beer}	{wine}	4	0.66
{wine}	{beer}	4	0.82

(b)

Fig. 4. (a) The `frequentPatterns` table and (b) the `rules` table which are the result of the evaluation of rules in Example 19.

$\mathcal{C}_M$ **constraints that are monotone but not succinct:** Monotone constraints, are exploited to reduce the input data and to prune the search space as described in [4, 5].

$\mathcal{C}_{AMS}$ **constraints that are both antimonotone and succinct:** constraints which are both antimonotone and succinct, can be satisfied before any mining takes place, by reducing the set of candidates 1-itemsets, to those items which satisfy such constraints as described in [11]. However, we can exploit them also in a data reduction fashion [4, 5].

$\mathcal{C}_{MS}$ **constraints that are both monotone and succinct:** are exploited as monotone constraints, by using them to reduce the data and the search space, and as succinct constraint in the candidate generation procedure, in the style of [11].

$\mathcal{C}_H$ **hard constraints:** Hard constraints, i.e. constraints which are neither antimonotone, nor monotone, nor succinct, are used inducing weaker constraints which exhibits some nice properties that allows pushing in the computation, and then they are checked at the end of the computation. In particular in our small language, as hard constraints, we have only the family of constraints based on the `avg` aggregate. The constraint $avg\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$ induces the weaker constraint $min\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$, which is both monotone and succinct and therefore it will be pushed in the computation to reduce the data and the search space. Analogously the constraint $avg\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$ induces the weaker, succinct and monotone, constraint $max\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$. However at the end of the computation, frequent itemsets which satisfy all constraints will be checked against the `avg` based constraint.

A particular kind of constraint is the cardinality constraint: when it is in the form $|s| \geq m$, it is a monotone constraint, and as that is treated. When it is in the form $|s| \leq m$, is an antimonotone constraint, but it differs from the other antimonotone constraints since it does not need to be checked for all itemsets, since the computation itself is level-wise (at iteration k we count frequency of k-itemsets). Thus this constraint just imposes a stopping condition (level m) to the level-wise computation. We create a special class for this constraint: $\mathcal{C}_{Stop}$.

Following the above consideration we define a new operator, that puts each constraint in the query in the proper class. All these classes, once populated, will feed the mining algorithm as parameters.

Definition 20 (Constraints manager). Given a conjunction of constraints $\mathcal{C}$, we define the constraints manager operator, $\mathcal{CM}(\mathcal{C})$ as the operator which takes all constraints in $\mathcal{C}$, and put them in one or more class of constraints, as described in Figure 5.

Once the rule parser operator and the constraint manager have prepared the computation, an optimized mining operator $\mathcal{M}(MV, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C}_{AM}, \mathcal{C}_M, \mathcal{C}_{AMS}, \mathcal{C}_{MS}, \mathcal{C}_H, \mathcal{C}_{Stop})$ will start the level-wise Apriori-like computation using all the constraints available in order to reduce as much as possible the input data and the search space. The details about how $\mathcal{M}$ is implemented, as well as the pseudo-code are reported in [2] and in Appendix A. The optimized operational semantics for our inductive rules is defined as in Figure 6.

$\mathbf{C_{ID}}$	**Constraint** C	$\mathcal{CM}(C)$
1	$s \supseteq \{a_1, \ldots, a_n\}$	$\mathcal{C}_{AMS}$
2	$s \subseteq \{a_1, \ldots, a_n\}$	$\mathcal{C}_{MS}$
3	$\lvert s \rvert \geq m$	$\mathcal{C}_{M}$
4	$\lvert s \rvert \leq m$	$\mathcal{C}_{Stop}$
5	$\lvert s \rvert = m$	$\mathcal{C}_{M}, \mathcal{C}_{Stop}$
6	$s.d \supseteq S$	$\mathcal{C}_{MS}$
7	$s.d \subseteq S$	$\mathcal{C}_{AMS}$
8	$min\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$	$\mathcal{C}_{MS}$
9	$min\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$	$\mathcal{C}_{AMS}$
10	$min\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$	$\mathcal{C}_{AMS}, \mathcal{C}_{MS}$
11	$max\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$	$\mathcal{C}_{AMS}$
12	$max\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$	$\mathcal{C}_{MS}$
13	$max\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$	$\mathcal{C}_{AMS}, \mathcal{C}_{MS}$

$\mathbf{C_{ID}}$	**Constraint** C	$\mathcal{CM}(C)$
14	$sum\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$	$\mathcal{C}_{AM}$
15	$sum\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$	$\mathcal{C}_{M}$
16	$sum\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$	$\mathcal{C}_{AM}, \mathcal{C}_{M}$
17	$count\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$	$\mathcal{C}_{AM}$
18	$count\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$	$\mathcal{C}_{M}$
19	$count\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$	$\mathcal{C}_{AM}, \mathcal{C}_{M}$
20	$avg\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$	$\mathcal{C}_{H}, \mapsto 8$
21	$avg\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$	$\mathcal{C}_{H}, \mapsto 12$
22	$avg\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$	$\mathcal{C}_{H}, \mapsto 8, 12$
23	$range\{d \mid i \in S \wedge P(i, \ldots, d)\} \leq m$	$\mathcal{C}_{AM}$
24	$range\{d \mid i \in S \wedge P(i, \ldots, d)\} \geq m$	$\mathcal{C}_{M}$
25	$range\{d \mid i \in S \wedge P(i, \ldots, d)\} = m$	$\mathcal{C}_{AM}, \mathcal{C}_{M}$

Fig. 5. Constraints Manager Table.

Optimized operational semantics of an inductive rule r

1. **if** $\mathcal{SC}(r)$
2. **then**
3. $\mathcal{RP}(r) = freq_{\langle \mathcal{V}, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C} \rangle}(t_1, \ldots, t_n)$;
4. $\mathcal{Q}(\mathcal{V}) = MV$;
5. $\mathcal{CM}(\mathcal{C}) \mapsto \mathcal{C}_{AM}, \mathcal{C}_{M}, \mathcal{C}_{AMS}, \mathcal{C}_{MS}, \mathcal{C}_{H}, \mathcal{C}_{Stop}$
6. $\mathcal{M}(MV, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C}_{AM}, \mathcal{C}_{M}, \mathcal{C}_{AMS}, \mathcal{C}_{MS}, \mathcal{C}_{H}, \mathcal{C}_{Stop}) = S$;
7. **for all** $s \in S$ **return**$(s, t_1, \ldots, t_n)$;
8. **else return** *unsafe rule*

Fig. 6. Optimized operational semantics of an inductive rule r.

6 Conclusions

This paper has provided a formalization of frequent pattern queries over relational databases. This formalization has permitted to define an inductive language for frequent pattern queries, which is simple enough to be highly optimized and expressive enough to cover the most of interesting queries. Following the recente results in constrained frequent pattern mining algorithms, we have defined an optimized constraint-pushing operative semantics for the queries of the proposed language.

There are some issues left uncovered by this research, for which further investigations are needed. An important issue is how to tightly integrate the inductive engine (optimized algorithms for inductive query) with the deductive DBMS. This issue is strictly connected with many other open problems ranging from how to store and index frequent pattern query results to how to reuse them for incremental query refinement. We believe that all this class of problems must be attacked together with an unifying approach.

References

1. R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 487–499, Santiago, Chile, 1994.
2. F. Bonchi. *Frequent Pattern Queries: Language and Optimizations*. PhD thesis, Computer Science Department, University of Pisa, Italy, December 2003.
3. F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. Adaptive Constraint Pushing in frequent pattern mining. In *Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD03)*, 2003.
4. F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. ExAMiner: Optimized level-wise frequent pattern mining with monotone constraints. In *Proceedings of the Third IEEE International Conference on Data Mining (ICDM'03)*, 2003.
5. F. Bonchi, F. Giannotti, A. Mazzanti, and D. Pedreschi. Exante: Anticipated data reduction in constrained pattern mining. In *Proceedings of the 7th European Conference on Principles and Practice of Knowledge Discovery in Databases (PKDD03)*, 2003.
6. F. Giannotti and G. Manco. Querying Inductive Databases via Logic-Based User-Defined Aggregates. In J. Rauch and J. Zitkov, editors, *Procs. of the European Conference on Principles and Practices of Knowledge Discovery in Databases*, number 1704 in Lecture Notes on Artificial Intelligence, pages 125–135, September 1999.
7. F. Giannotti, G. Manco, M. Nanni, and D. Pedreschi. Nondeterministic, Nonmonotonic Logic Databases. *IEEE Transactions on Knowledge and Data Engineering*. 32(2):165-189, October 2001.
8. B. Goethals. *Efficient Frequent Pattern Mining*. PhD thesis, transnational University of Limburg, Diepenbeek, Belgium, December 2002.
9. J. Han, L. V. S. Lakshmanan, and R. T. Ng. Constraint-based, multidimensional data mining. *Computer*, 32(8):46–50, 1999.
10. H. Mannila and H. Toivonen. Levelwise Search and Border of Theories in Knowledge Discovery. *Data Mining and Knowledge Discovery*, 3:241–258, 1997.
11. R. T. Ng, L. V. S. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained associations rules. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-98).*

A Appendix: The Optimized Mining Operator $\mathcal{M}$

In Figure 7 the pseudo-code for the mining operator $\mathcal{M}$ is reported. Knowledge of the algorithms Apriori [1], ExAnte [5], ExAMiner [4] and CAP [11] is required in order to understand the pseudo-code. The mining operator performs an ExAnte [5] preprocessing at level 1 (lines from 1 to 18), where:

- both $\mathcal{C}_M$ and $\mathcal{C}_{MS}$ are exploited in order to μ-reduce the input database (lines 3 and 11);
- $\mathcal{C}_{AM}$ and $\mathcal{C}_{AMS}$ are exploited in order to α-reduce but only at the first round (line 6): after this reduction they will not be checked again at level 1, since they can not shrink.

Then it starts an Apriori generate and test computation where:

- $\mathcal{C}_{Stop}$ is used as additional stopping condition (line 22);

– the **generate_apriori** [1] (lines 20 and 24) procedure exploits succinct monotone constraints, as defined in [11];
– the **count** procedure is substituted by an ExAMiner **count**&**reduce** (line 23) procedure as defined in [4]: it uses $\mathcal{C}_M$, $\mathcal{C}_{AM}$ and $\mathcal{C}_{MS}$ to reduce the dataset;
– the final test (line 27), check L_i for satisfaction of $\mathcal{C}_M$ and $\mathcal{C}_H$.

Note that in the final test of constraints, we do not need to check $\mathcal{C}_{MS}$, since their satisfaction is assured by the candidate generation procedure. The mining operator $\mathcal{M}$ defined above, corresponds to an $\mathbf{ExAMiner_1}$ computation, since it loops only at the first level, and then goes on strictly level-wise using the **count**&**reduce** procedure typical of ExAMiner. However, it is enriched by pushing all possible constraints in the proper step of the computation.

Mining operator: $\mathcal{M}(MV, \mathcal{T}, \mathcal{I}, \sigma, \mathcal{C}_{AM}, \mathcal{C}_M, \mathcal{C}_{AMS}, \mathcal{C}_{MS}, \mathcal{C}_H, \mathcal{C}_{Stop})$

1. $Items := \emptyset$;
2. **forall** transactions $\langle \mathcal{T}, \{\mathcal{I}\} \rangle$ in MV **do**
3.   **if** $\{\mathcal{I}\} \in Th(\mathcal{C}_M \cap \mathcal{C}_{MS})$
4.   **then forall** items i in $\{\mathcal{I}\}$ **do**
5. $i.count{+}{+}$;
6.   **if** $i.count == \sigma$ **and** $\{i\} \in Th(\mathcal{C}_{AM} \cap \mathcal{C}_{AMS})$
7.   **then** $Items := Items \cup i$;
8. $old_number_interesting_items := |Items|$;
9. **while** $|Items| < old_number_interesting_items$ **do**
10. $MV := \alpha[MV]_{\mathcal{C}_{freq}}$;
11. $MV := \mu[MV]_{\mathcal{C}_M \cap \mathcal{C}_{MS}}$;
12. $old_number_interesting_items := |Items|$;
13. $Items := \emptyset$;
14.   **forall** transactions $\langle \mathcal{T}, \{\mathcal{I}\} \rangle$ in MV **do**
15.   **forall** items i in $\{\mathcal{I}\}$ **do**
16. $i.count + +$;
17.   **if** $i.count == \sigma$ **then** $Items := Items \cup i$;
18. **end while**
19. $L_1 := Items$; $C_2 := generate_apriori(L_1, \mathcal{C}_{MS})$; $k := 2$;
20. **while** $C_k \neq \emptyset$ **and not** $\mathcal{C}_{Stop}$ **do**
21. $L_k := count\&reduce(\mathcal{C}_{freq} \cap \mathcal{C}_{AM}, C_k, MV, \mathcal{C}_M \cap \mathcal{C}_{MS})$;
22. $C_{k+1} := generate_apriori(L_k, \mathcal{C}_{MS})$;
23.   k++;
24. **end while**
25. **return** $\bigcup_{i=1}^{k-1} L_i \cap Th(\mathcal{C}_M) \cap Th(\mathcal{C}_H)$

Fig. 7. Pseudo-code of the mining operator $\mathcal{M}$.

Costruzione automatica di courseware in DyLOG

Laura Torasso

Dipartimento di Informatica – Università degli Studi di Torino
C.so Svizzera, 185 – I-10149 Torino (Italy)
torassol@di.unito.it
Società Reale Mutua di Assicurazioni
Servizio Informatico-S.I.Ge.A.
Via Corte d'Appello 11, Torino

Abstract. Nelle piattaforme di e-learning i corsi non possono seguire una struttura classica. I nuovi strumenti messi a disposizione dalla rete (chat, forum ipermedialità) impongono una rivisitazione della struttura dei contenuti e delle strategie didattiche utilizzate. In questo articolo si vuole descrivere un idea per aiutare i docenti nella definizione di un corso, inteso come materiale educativo e strategia didattica, avvalendosi delle capacità espressive e di ragionamento del paradigma logico.

Descrizione del problema

Questi ultimi anni hanno assistito ad un fiorire di piattaforme di e-learning, mirate all'utilizzo di materiale didattico online, indicato col nome di courseware, unità didattica o learning object. La fase più delicata e importante nella progettazione di un corso, soprattutto se a distanza, è la scelta della *strategia didattica* da adottare. Oggigiorno psicologia e pedagogia offrono ai formatori un panorama di modelli di insegnamento (in aula, a distanza e sul lavoro) molto più vasto che in passato, nei quali si pone particolare attenzione alla figura del discente, che passa dal ruolo passivo di *uditore* a quello attivo di *promotore* del proprio apprendimento. Questa transizione è facilitata dalla diffusione dell'accesso ad Internet. Infatti, se nel modello classico gli argomenti (documenti, dispense) sono presentati in modo sequenziale ed eventuali esempi ed esercitazioni pratiche non hanno alcuna ricaduta sulla sequenza di presentazione del materiale, legata alla scaletta del corso, l'utilizzo del Web come veicolo consente al discente un approccio "esplorativo", in quanto egli può navigare attraverso i contenuti proposti, soffermandosi su quelli che ritiene più interessanti. In generale una buona strategia didattica deve consentire un certo grado di esplorazione, vincolando però le possibilità del discente, il quale deve essere guidato in un percorso didattico fruttuoso e possibilmente personalizzato, nel quale i risultati di test ed esercizi sono tenuti in considerazione nella scelta del materiale da proporre.

Costruire una strategia didattica significa quindi specificare come le unità (più tecnicamente: i *learning object*) devono essere presentate, dal punto di vista grafico ma anche e soprattutto secondo quale schema o ordine.

Se si riuscisse a rappresentare in modo modulare learning object e strategie didattiche, entrambi potrebbero essere riutilizzati nella costruzione di nuovi

corsi. In questo contesto risulta interessante da un lato definire formalismi e standard per la specifica di meta-informazioni riguardanti i learning object (ad es. prerequisiti ed obiettivi), dall'altro sviluppare degli strumenti che aiutino il docente a comporre il materiale da fruire online per un certo corso, applicando la strategia didattica da questi ritenuta più opportuna ad un insieme di learning object disponibili ed eventualmente realizzati da terzi. In generale, sarebbe addirittura possibile costruire learning object ad hoc per un certo studente, basandosi sulle preferenze e caratteristiche di quest'ultimo (presenza di esercizi aggiuntivi, necessità di approfondimenti, eccetera).

In questo articolo si vuole illustrare in breve come DyLOG, un linguaggio logico per ragionare su azioni e cambiamento, già utilizzato in applicazioni Web-based educational [3, 4, 2], possa essere utilizzato sia per rappresentare la conoscenza richiesta per espletare il compito descritto sia per costruire un agente razionale che, partendo dalla descrizione degli obiettivi didattici di un corso, dalla descrizione di un insieme di learning object, costituenti i contenuti di un repository didattico, e dalla specifica astratta di una strategia didattica, sia in grado di assemblare le specifiche per la presentazione di un corso in una piattaforma di e-learning.

Learning Object e strategie didattiche come azioni

Un sistema intelligente ragiona sul proprio comportamento e adotta una certa strategia sulla base di uno stato *mentale interno*. Il sistema che intendiamo realizzare deve poter ragionare sui learning object (LO nel seguito) per riuscire a comporre dei moduli didattici. Occorre quindi fornire una descrizione dichiarativa dei LO. Una scelta piuttosto immediata è descrivere tali unità come azioni atomiche, definendone precondizioni all'esecuzione ed effetti in termini di competenze da acquisire e competenze offerte. Nel seguito "ereditarietà" è un LO, rappresentato in simil-DyLOG, inerente il concetto di ereditarietà in Java. L'acquisizione di tale concetto presuppone la conoscenza dei concetti di oggetto e interfaccia -knows(oggetti) & knows(interfacce)- e comporta l'acquisizione (**causes**) di competenze su binding dinamico, up- e down-casting:

learning_object(ereditarieta) **causes**
  knows(dinamic_binding) & knows(down_casting) & knows(up_casting)
  if knows(oggetti) & knows(interfacce).
learning_object(introduzione) **causes**
  knows(oggetti) & knows(costruttori) & knows(interfacce).

strategy(topic) **is**
  user(initial_test, topic) & add_los(topic) & user(score_test, topic).

Nella definizione di un corso di Java, il sistema automatico farà precedere questo LO da uno o più moduli che forniscono le competenze precondizione, ad es. "introduzione". Il repository potrebbe contenere diversi LO che offrono le stesse competenze ma caratterizzati in modo diverso. L'uso dell'uno o dell'altro consentirà di meglio adattare il materiale prodotto alle esigenze dell'utente. Possiamo infine interpretare le *strategie didattiche* come azioni complesse, che definiscono

schemi generici di possibili corsi, svincolati dai contenuti e tali da regolarne la struttura. Nell'esempio è descritto una strategia didattica (strategy) su un generico argomento (topic), che produce corsi costituiti da un pre-test, seguito da uno o più learning object (estratti dal repository da make_los) ed un test finale.

Lavori in corso

In questo articolo abbiamo mostrato un possibile uso del linguaggio DyLOG nel campo dei sistemi per l'e-learning. Il nostro lavoro ora è rivolto allo sviluppo di un tutor virtuale in grado di sostenere il docente nella definizione di un courseware: tramite un agente intelligente vogliamo organizzare i LO seguendo strategie didattiche personalizzabili. A tal fine i LO devono essere arricchiti da una descrizione semantica del loro contenuto e tipo. Definiti i LO e stabilite le dipendenze tra i concetti, per costruire un courseware basterà indicare le competenze che questo deve fornire e la strategia didattica da seguire. L'agente, ragionando sulle dipendenze di causa ed effetto e sulle preferenze espresse, produrrà un corso. Poiché la fruizione di un corso è vincolato dalla tecnologia usata dal sistema di apprendimento, il nostro agente descriverà il courseware prodotto secondo lo standard SCORM [1]. Tale standard si è dimostrato una delle proposte di maggior successo degli ultimi anni nella descrizione del'uso e il riuso dei LO. Caratteristica interessante della versione più recente di SCORM è che consente di esprimere la sequenzializzazione e la navigabilità dei learning object tramite semplici regole di verifica di stato, basate sul grado di completamento misurato durante l'esecuzione dell'attività. Tali regole derivano direttamente dalla strategia didattica adottata.

Ringraziamenti

Per il loro sostegno e aiuto ringrazio Matteo Baldoni e Cristina Baroglio del Dipartimento di Informatica dell'Università di Torino.

References

1. Scorm specifications. Available at `http://www.adlnet.org`.
2. M. Baldoni, C. Baroglio, B. Demo, V. Patti, and L. Torasso. E-learning by doing, an approach based on techniques for reasoning about actions. In *Proc. of the Workshop on Artificial Intelligence and E-learning, AIIA2003, Selected Papers*, pages 43–55, Pisa, Italy, 2003.
3. M. Baldoni, C. Baroglio, and V. Patti. Applying logic inference techniques for gaining flexibility and adaptivity in tutoring systems. In C. Stephanidis, editor, *Proceedings of the 10th Int. Conf. on Human-Computer Interaction*, Crete, Grece, 2003. Lawrence Erlbaum Associates.
4. M. Baldoni, C. Baroglio, V. Patti, and L. Torasso. Using a rational agent in an adaptive web-based tutoring system. In *Proc. of the Workshop on Adaptive Systems for Web-Based Education, AH2002, Selected Papers*, pages 43–55, Malaga, Spain, 2002.

Improving efficiency of recursive theory learning

Antonio Varlaro, Margherita Berardi, Donato Malerba

Dipartimento di Informatica – Università degli Studi di Bari
via Orabona 4 - 70126 Bari
{varlaro, berardi, malerba}@di.uniba.it

Abstract. Inductive learning of recursive logical theories from a set of examples is a complex task characterized by three important issues, namely the adoption of a generality order stronger than θ-subsumption, the non-monotonicity of the consistency property, and the automated discovery of dependencies between target predicates. Solutions implemented in the learning system ATRE are briefly reported in the paper. Moreover, efficiency problems of the learning strategy are illustrated and two caching strategies, one for the clause generation phase and one for the clause evaluation phase, are described. The effectiveness of the proposed caching strategies has been tested on the document processing domain. Experimental results are discussed and conclusions are drawn.

1. Introduction

Inductive Logic Programming (ILP) has evolved from previous research in Machine Learning, Logic Programming, and Inductive Program Synthesis. Like Machine Learning, it deals with the induction of concepts from observations (examples) and the synthesis of new knowledge from experience. Its peculiarity is the use of computational logic as the representation mechanism for concept definitions and observations. Typically, the output of an ILP system is a logical theory expressed as a set of definite clauses, which logically entail all positive examples and no negative example. Therefore, each concept definition corresponds to a predicate definition and a concept learning problem is reformulated as a predicate learning problem.

Learning a single predicate definition from a set of positive and negative examples is a classical problem in ILP. In this paper we are interested in the more complex case of learning multiple predicate definitions, provided that both positive and negative examples of each concept/predicate to be learned are available. Complexity stems from the fact that the learned predicates may also occur in the antecedents of the learned clauses, that is, the learned predicate definitions may be interrelated and depend on one another, either hierarchically or involving some kind of mutual recursion. For instance, to learn the definitions of odd and even numbers, a multiple predicate learning system will be provided with positive and negative examples of both odd and even numbers, and may generate the following recursive logical theory:

$$odd(X) \leftarrow succ(Y,X), even(Y)$$
$$even(X) \leftarrow succ(Y,X), odd(Y)$$

$$even(X) \leftarrow zero(X)$$

where the definitions of *odd* and *even* are interdependent. This example shows that the problem of learning multiple predicate definitions is equivalent, in its most general formulation, to the problem of learning recursive logical theories.

There has been considerable debate on the actual usefulness of learning recursive logical theories in knowledge acquisition and discovery applications. It is a common opinion that very few real-life concepts seem to have recursive definitions, rare examples being "ancestor" and natural language [2, 10]. Despite this scepticism, in the literature it is possible to find several ILP applications in which recursion has proved helpful [7]. Moreover, many ILP researchers have shown some interest in multiple predicate learning [6], which presents the same difficulty of recursive theory learning in its most general formulation.

To formulate the recursive theory learning problem and then to explain its main theoretical issues, some basic definitions are given below.

Generally, every logical theory T can be associated with a directed graph $\gamma(T)=\langle N,E\rangle$, called the *dependency graph* of T, in which (i) each predicate of T is a node in N and (ii) there is an arc in E directed from a node a to a node b, iff there exists a clause C in T, such that a and b are the predicates of a literal occurring in the head and in the body of C, respectively.

A dependency graph allows representing the predicate dependencies of T, where a *predicate dependency* is defined as follows:

Definition 1 (predicate dependency). A predicate p depends on a predicate q in a theory T iff (i) there exists a clause C for p in T such that q occurs in the body of C; or (ii) there exists a clause C for p in T with some predicate r in the body of C that depends on q.

Definition 2 (recursive theory). A logical theory T is *recursive* if the dependency graph $\gamma(T)$ contains at least one cycle.

In *simple* recursive theories all cycles in the dependency graph go from a predicate p into p itself, that is, simple recursive theories may contain recursive clauses, but cannot express mutual recursion.

Definition 3 (predicate definition). Let T be a logical theory and p a predicate symbol. Then the *definition* of p in T is the set of clauses in T that have p in their head. Henceforth, $\delta(T)$ will denote the set of predicates defined in T and $\pi(T)$ will denote the set of predicates occurring in T, then $\delta(T)\subseteq\pi(T)$.

In a quite general formulation, the recursive theory learning task can be defined as follows:

Given

- A set of *target* predicates $p_1, p_2, \ldots, p_r$ to be learned
- A set of positive (negative) examples E_i^+ (E_i^-) for each predicate p_i, $1\leq i\leq r$
- A background theory BK
- A language of hypotheses $\mathcal{L}_H$ that defines the space of hypotheses S_H

Find

a (possibly recursive) logical theory $T\in S_H$ defining the predicates $p_1, p_2, \ldots, p_r$ (that is, $\delta(T)=\{p_1, p_2, \ldots, p_r\}$) such that for each i, $1\leq i\leq r$, $BK\cup T \models E_i^+$ (*completeness* property) and $BK\cup T \not\models E_i^-$ (*consistency* property).

Three important issues characterize recursive theory learning. First, the generality order typically used in ILP, namely θ-subsumption [13], is not sufficient to guarantee the completeness and consistency of learned definitions, with respect to logical entailment [12]. Therefore, it is necessary to consider a stronger generality order, which is consistent with the logical entailment for the class of recursive logical theories we take into account.

Second, whenever two individual clauses are consistent in the data, their conjunction need not be consistent in the same data [5]. This is called the non-monotonicity property of the normal ILP setting, since it states that adding new clauses to a theory T does not preserve consistency. Indeed, adding definite clauses to a definite program enlarges its least Herbrand model (LHM), which may then cover negative examples as well. Because of this non-monotonicity property, learning a recursive theory one clause at a time is not straightforward.

Third, when multiple predicate definitions have to be learned, it is crucial to discover dependencies between predicates. Therefore, the classical learning strategy that focuses on a predicate definition at a time is not appropriate.

To overcome these problems a new approach to the learning of multiple dependent concepts has been proposed in [8] and implemented in the learning system ATRE (www.di.uniba.it/~malerba/software/atre). This approach differs from related works for at least one of the following three aspects: the learning strategy, the generalization model, and the strategy to recover the consistency property of the learned theory when a new clause is added.

The paper synthesizes and extends the work presented in [8]. In particular, it presents a brief overview of solutions proposed and implemented in ATRE to the three main issues above. Evolutions of the search strategy are also reported. More precisely, two new issues regarding the search space exploration are faced, one concerning search bias definition in order to allow the user to guide the search space exploration according to his/her preference, and the other one concerning efficiency problems due to the computational complexity of the search space. Some solutions have been proposed and implemented in a new version of the system ATRE.

The paper is organized as follows. Section 2 illustrates issues and solutions related to the recursive theory learning. Section 3 introduces efficiency problems and presents optimization approaches adopted in ATRE. Section 4 illustrates the application of ATRE on real-world documents and presents results on efficiency gain. Finally, in Section 5 some conclusions are drawn.

2. Issues and solutions

2.1 The generality order

As explained above, in recursive theory learning it is necessary to consider a generality order that is consistent with the logical entailment for the class of recursive logical theories. A generality order (or *generalization model*) provides a basis for organizing the search space and is essential to understand how the search strategy proceeds. The main problem with the well-known θ-subsumption is that the objects

of comparison are two clauses, say *C* and *D*, and no additional source of knowledge (e.g., a theory *T*) is considered. For instance, with reference to the previous example on *odd* and *even* predicates, the clause:

C: *odd(X)* ← *succ(Y,X)*, *even(Y)*

logically entails, and hence can be correctly considered more general than

D: *odd(3)* ← *succ(0,1)*, *succ(1,2)*, *succ(2,3)*, *even(0)*

only if we take into account the theory

T: *even(A)* ← *succ(B,A)*, *odd(B)*
even(C) ← *zero(C)*

Therefore, we are only interested in those generality orders that compare two clauses relatively to a given theory *T*, such as Buntine's *generalized subsumption* [3] and Plotkin's notion of *relative generalization* [13, 14].

Informally, generalized subsumption ($\leq_T$) requires that the heads of *C* and *D* refer to the same predicate, and that the body of *D* can be used, together with the background theory *T*, to entail the body of *C*. Unfortunately, generalized subsumption is too weak for recursive theories, because in some cases, given two clauses C and D, it may happen that $T\cup\{C\}\models D$ holds but it can not be concluded that $C\leq_T D$.

Plotkin's notion of *relative generalization* [13, 14] was originally proposed for a theory T of unit clauses. Buntine [3] reports an extension of relative generalization to the case of a theory T composed of definite clauses (not necessarily of unit clauses)

Definition 4 (relative generalization). Let *C* and *D* be two definite clauses. *C* is *more general than D* under relative generalization, with respect to a theory *T*, if a substitution θ exists such that $T \models \forall(C\theta \Rightarrow D)$.

The following theorem holds for this extended notion of relative generalization:

Theorem 1. Let *C* and *D* be two definite clauses and T a logical theory. *C* is *more general than D* under relative generalization, with respect to a theory *T*, if and only if C occurs at most once in some refutation demonstrating $T \models \forall(C \Rightarrow D)$.

However, this extended notion of relative generalization is still inadequate. From one side, it is still weak. Indeed, if we consider the clauses and the theory reported in the example above, it is clear that a refutation demonstrating $T \models \forall(C \Rightarrow D)$ involves twice the clause C to prove both *odd(1)* and *odd(3)*.

Malerba [8] has defined the following generalization order, which proved suitable for recursive theories.

Definition 5 (generalized implication). Let *C* and *D* be two definite clauses. *C* is *more general than D* under generalized implication, with respect to a theory *T*, denoted as $C\leq_{T\Rightarrow}D$, if a substitution θ exists such that $head(C)\theta = head(D)$ and $T \models \forall(C \Rightarrow D)$.

Decidability of the generalized implication test is guaranteed in the case of Datalog clauses [4]. In fact, the restriction to function-free clauses is common in ILP systems, such as ATRE, which remove function symbols from clauses and put them in the background knowledge by techniques such as flattening [15].

2.2 The non-monotonicity property

It is noteworthy that generalized implication compares two definite clauses for generalization. This means that the search space structured by this generality order is the space of definite clauses. A recursive logical theory is generally composed of several clauses, therefore the learning strategy must search for one clause at a time. More precisely, a recursive theory T is built step by step, starting from an empty theory T_0, and adding a new clause at each step. In this way we get a sequence of theories

$$T_0 = \varnothing, T_1, \ldots, T_i, T_{i+1}, \ldots, T_n = T,$$

such that $T_{i+1} = T_i \cup \{C\}$ for some clause C. If we denote by LHM(T_i) the least Herbrand model of a theory T_i, the stepwise construction of theories entails that LHM(T_i) $\subseteq$ LHM(T_{i+1}), for each $i \in \{0, 1, \ldots, n\text{-}1\}$, since the addition of a clause to a theory can only augment the LHM. Henceforth, we will assume that both positive and negative examples of predicates to be learned are represented as *ground atoms* with a + or - label. Therefore, examples may or may not be elements of the models LHM(T_i). Let $pos(LHM(T_i))$ and $neg(LHM(T_i))$ be the number of positive and negative examples in LHM(T_i), respectively. If we guarantee the following two conditions:

1. $pos(LHM(T_i)) < pos(LHM(T_{i+1}))$ for each $i \in \{0, 1, \ldots, n\text{-}1\}$, and
2. $neg(LHM(T_i)) = 0$ for each $i \in \{0, 1, \ldots, n\}$,

then after a finite number of steps a theory T, which is complete and consistent, is built. This learning strategy is known as *sequential covering* (or *separate-and-conquer*) [9].

In order to guarantee the first of the two conditions it is possible to proceed as follows. First, a positive example e^+ of a predicate p to be learned is selected, such that e^+ is not in LHM(T_i). The example e^+ is called *seed*. Then the space of definite clauses more general than e^+ is explored, looking for a clause C, if any, such that neg(LHM($T_i \cup \{C\}$)) = $\varnothing$. In this way we guarantee that the second condition above holds as well. When found, C is added to T_i giving T_{i+1}. If some positive examples are not included in LHM(T_{i+1}) then a new seed is selected and the process is repeated.

The second condition is more difficult to guarantee because of the second issue presented in the introduction, namely, the non-monotonicity property. Algorithmic implications of this property may be effectively illustrated by means of an example. Consider the problem of learning the definitions of ancestor and father from a complete set of positive and negative examples. Suppose that the following recursive theory T_2 has been learned at the second step:

C_1: *ancestor*(*X,Y*) ← *parent(X,Y)*

C_2: *father*(*Z,W*) ← *ancestor(Z,W), male(Z)*

Note that T_2 is consistent but still incomplete. Thus a new clause will be generated at the third step of the sequential-covering strategy. It may happen that the generated clause is the following:

C: *ancestor*(*A,B*) ← *parent(A,D),ancestor(D,B)*

which is consistent given T_2, but when added to the recursive theory, it makes clause C_2 inconsistent.

There are several ways to remove such inconsistency by revising the learned theory. Nienhuys-Cheng and de Wolf [11] describe a complete method of

specializing a logic theory with respect to sets of positive and negative examples. The method is based upon unfolding, clause deletion and subsumption. These operations are not applied to the last clause added to the theory, but may involve any clause of the inconsistent theory. As a result, clauses learned in the first inductive steps could be totally changed or even removed. This theory revision approach, however, is not coherent with the stepwise construction of the theory T presented above, since it re-opens the whole question of the validity of clauses added in the previous steps. An alternative approach consists of simple syntactic changes in the theory, which eventually creates new *layers* in a logical theory, just as the stratification of a normal program creates new strata [1].

More precisely, a layering of a theory T is a partition of the clauses in T into n disjoint sets of clauses or *layers* T^i such that $LHM(T)= LHM(LHM(\cup_{j=0,\ldots,n-2} T^j)\cup T^{n-1})$, that is, $LHM(T)$ can be computed by iteratively applying the immediate consequence operator to T^i, starting from the interpretation $LHM(\cup_{j=0,\ldots,i-1} T^j)$, for each $i\in\{1, \ldots,n\}$. In [8] an efficient method for the computation of a layering is reported. It is based on the concept of collapsed dependency graph and returns a unique layering for a given logical theory T. The layering of a theory provides a semi-naive way of computing the generalized implication test presented above and provides a solution to the problem of consistency recovering when the addition of a clause makes the theory inconsistent.

Theorem. Let $T=T^0 \cup \ldots \cup T^i \cup \ldots \cup T^{n-1}$ be a consistent theory partitioned into n layers, and C be a definite clause whose addition to the theory T makes a clause in layer T^i inconsistent. Let $p\in\{p_1, p_2, \ldots, p_r\}$ be the predicate in the head of C. Let T'' be a theory obtained from T by substituting all occurrences of p in T with a new predicate symbol, p', and $T'=T''\cup\{p(t_1, \ldots, t_n) \leftarrow p'(t_1, \ldots, t_n)\}\cup\{C\}$. Then T' is consistent and $LHM(T) \subseteq LHM(T') \setminus \{p(t_1, \ldots, t_n) \leftarrow p'(t_1, \ldots, t_n)\}$.

In short, the new theory T' obtained by renaming the predicate p with a new predicate name p' before adding C is consistent and keeps the original coverage of T. This introduces a first variation of the classical separate-and-conquer strategy sketched above, since the addition of a locally consistent clause C generated in the conquer stage is preceded by a global consistency check. If the result is negative, the partially learned theory is first restructured, and then two clauses, $p(t_1, \ldots, t_n) \leftarrow p'(t_1, \ldots, t_n)$ and C, are added. For instance, in the example above the result will be:

C_1': *ancestor'* $(X,Y) \leftarrow parent(X,Y)$
C_2': *father*$(Z,W) \leftarrow ancestor'(Z,W), male(Z)$
ancestor$(U,V) \leftarrow ancestor'(U,V)$
C: *ancestor*$(A,B) \leftarrow parent(A,D), ancestor(D,B)$

It is noteworthy that, in the proposed approach to consistency recovery, new predicates are invented, which aim to accommodate previously acquired knowledge (theory) with the currently generated hypothesis (clause).

2.3 Discovering dependencies between predicates

The third and last issue to deal with is the automated discovery of dependencies between target predicates $p_1, p_2, \ldots, p_r$. A solution to this problem is based on another variant of the separate-and-conquer learning strategy. Traditionally, this strategy is adopted by single predicate learning systems that generate clauses with the same predicate in the head at each step. In multiple predicate learning (or recursive theory learning) clauses generated at each step may have different predicates in their heads. In addition, the body of the clause generated at the *i*-th step may include all target predicates $p_1, p_2, \ldots, p_r$ for which at least a clause has been added to the partially learned theory in previous steps. In this way, dependencies between target predicates can be generated.

Obviously, the order in which clauses of distinct predicate definitions have to be generated is not known in advance. This means that it is necessary to generate clauses with different predicates in the head and then to pick one of them at the end of each step of the separate-and-conquer strategy. Since the generation of a clause depends on the chosen seed, several seeds have to be chosen such that at least one seed per incomplete predicate definition is kept. Therefore, the search space is actually a forest of as many search-trees (called *specialization hierarchies*) as the number of chosen seeds. A directed arc from a node C to a node C' exists if C' is obtained from C by a single refinement step. Operatively, the (downward) refinement operator considered in this work adds a new literal to a clause.

The forest can be processed in parallel by as many concurrent tasks as the number of search-trees. Each task traverses the specialization hierarchy top-down (or general-to-specific), but synchronizes traversal with the other tasks at each level. Initially, some clauses at depth one in the forest are examined concurrently. Each task is actually free to adopt its own search strategy, and to decide which clauses are worth to be tested. If none of the tested clauses is consistent, clauses at depth two are considered. Search proceeds towards deeper and deeper levels of the specialization hierarchies until at least a user-defined number of consistent clauses is found. Task synchronization is performed after that all "relevant" clauses at the same depth have been examined. A supervisor task decides whether the search should carry on or not on the basis of the results returned by the concurrent tasks. When the search is stopped, the supervisor selects the "best" consistent clause according to the user's preference criterion. This strategy has the advantage that simpler consistent clauses are found first, independently of the predicates to be learned.[1] Moreover, the synchronization allows tasks to save much computational effort when the distribution of consistent clauses in the levels of the different search-trees is uneven. The parallel exploration of the specialization hierarchies for *odd* and *even* is shown in Fig. 1.

---

[1] Apparently, some problems might occur for those recursive definitions where the recursive clause is syntactically simpler than the base clause. However, the proposed strategy does not allow the discovery of the recursive clause until the base clause has been found, whatever its complexity is.

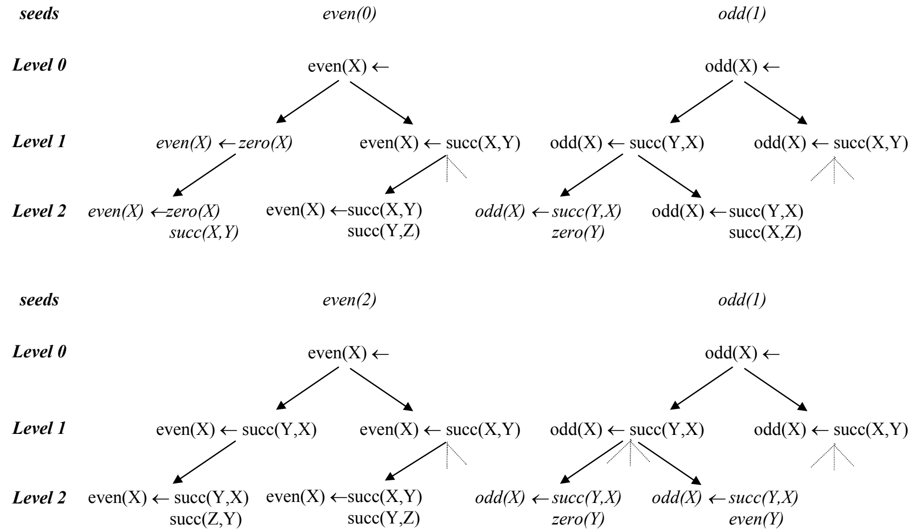


Fig. 1. Two steps (up and down) of the parallel search for the predicates *odd* and *even*. Consistent clauses are reported in italics.

2.4 Some refinements on the learning strategy

The learning strategy reported in previous section is quite general and there is room for several distinct implementations. In particular, the following three points have been left unspecified: 1) how seeds are selected; 2) what are the roots of specialization hierarchies; 3) what is the search strategy adopted by each task. In this section, solutions adopted in the last release of the learning system ATRE are illustrated.

Seed selection is a critical point. In the example of Fig. 1, if the search had started from *even(2)* and *odd(1)*, the first clause added to the theory would have been *odd(X) ← succ(Y,X), zero(Y)*, thus resulting in a less compact, though still correct, theory for odd and even numbers. Therefore, it is important to explore the specialization hierarchies of several seeds for each predicate. When training examples and background knowledge are represented either as sets of ground atoms (flattened representation) or as ground clauses, the number of candidate seeds can be very high, so the choice should be stochastic. The object-centered representation adopted by ATRE has the advantage of reducing the number of candidate seeds by partitioning the whole set of training examples *E* into *training objects*. The main assumption made in ATRE is that *each object contains examples explained by some base clauses of the underlying recursive theory*.[2] Therefore, by choosing as seeds *all* examples of different concepts represented in one training object, it is possible to induce some of the correct base clauses. Since in many learning problems the number of positive

---

[2] Problems caused by incomplete object descriptions violating the above assumption are not investigated in this work, since they require the application of *abductive* operators, which are not available in the current version of the system.

examples in an object is not very high, a parallel exploration of all candidate seeds is feasible. Mutually recursive concept definitions will be generated only after some base clauses have been added to the theory.

Seeds are chosen according to the textual order in which objects are input to ATRE. If a complete definition of the predicate p_j is not available yet at the i-th step of the separate-and-conquer search strategy, then there are still some uncovered positive examples of p_j. The first (seed) object O_k in the object list that contains uncovered examples of p_j is selected to generate seeds for p_j.

Generally, each specialization hierarchy is rooted in a unit clause, that is, a clause with an empty body. However, in some cases, the user has a clear idea of relevant properties that should appear in the body of the clauses and is even able to define the root of the specialization hierarchies. A *language bias* has been defined in ATRE to allow users to express constraints that should be satisfied by root clauses or by interesting clauses in the specialization hierarchy. In its current version, the language bias includes the following declarations:

$$starting_number_of_literals(p_i,N)$$
$$starting_clause(p_i,[L_1,L_2,...,L_N])$$

where p_i is a target predicate, N is a cardinal number, and $[L_1,L_2,...,L_N]$ represents a list of literals. In particular, the *starting_number_of_literals* declaration specifies the initial length of the root clause (at least N literals in the body), while the *starting_clause* declaration specifies a conjunctive constraint on the body of a root clause: all literals in the list $[L_1,L_2,...,L_N]$ must occur in the clause. Multiple *starting_clause* declarations for the same target predicate p_i specify alternative conjunctive constraints for the root clauses of specialization hierarchies associated to p_i. In addition, the following declaration:

$$starting_literal(p_i,[L_1,L_2,...,L_N])$$

specifies a disjunctive constraint at literal level for the body of root clauses. Literals are expressed as follows:

$$\text{f(decl-arg}_1, \ldots, \text{decl-arg}_n) = \text{Value}$$
$$\text{g(decl-arg}_1, \ldots, \text{decl-arg}_n) \in \text{Range}$$

where *decl-arg*'s are mode declarations for predicate arguments. Declarations are applicable only to variables and influence the way of generating variables. Two modes are available: *old* and *new*. The first mode means that the variable is an input variable, that is, it corresponds to a variable already occurring in the clause. The second mode means that the variable is a new one. Furthermore, values and ranges of predicates can be ground or not.

The third undefined point of the search strategy concerns the search strategy adopted by each task. ATRE applies a variant of the beam-search strategy. The system generates all candidate clauses at level $l+1$ starting from those filtered at level l in the specialization hierarchy. During task synchronization, which occurs level-by-level, the best m clauses are selected from those generated by all tasks. The user specifies the beam of the search, that is m, and a set of preference criteria for the selection of the best m clauses.

3. Improving efficiency in ATRE

In this section we present a novel caching strategy implemented in ATRE to overcome efficiency problems. Generally speaking, caching aims to save useful information that would be repeatedly recomputed otherwise, with a clear waste of time. In ATRE caching affects the two most computationally expensive phases of the learning process, namely the clause generation step and the clause evaluation step.

3.1 Caching for clause generation

The learning strategy sketched in Section 2.3 presents a large margin for optimization. One of the reasons is that every time a clause is added to the partially learned theory, the specialization hierarchies are reconstructed for a new set of seeds, which may intersect the set of seeds explored in the previous step. Therefore, it is possible that the system explores the same specialization hierarchies several times, since it has no memory of the work done in previous steps. This is particularly evident when concepts to learn are neither recursively definable nor mutually dependent. Caching the specialization hierarchies explored at the *i*-th step of the separate-and-conquer strategy and reusing part of them at the *(i+1)*-th step, seems to be a good strategy to decrease the learning time while keeping memory usage under acceptable limits.

First of all, we observe that a necessary condition for reusing a specialization hierarchy between two subsequent learning steps is that the associated seed remains the same. This means that if the seed of a specialization hierarchy is no longer considered at the *(i+1)*-th step, then the corresponding clauses cached at the *i*-th step can be discarded.

However, even in the case of same seed, not all the clauses of the specialization hierarchy will be actually useful. For instance, the cached copies of a clause *C* added to T_i can be removed from all specialization hierarchies including it. Moreover, all clauses that cover only positive examples already covered by *C* can be dropped, according to the separate-and-conquer learning strategy. These examples explain why a cached specialization hierarchy has to be pruned before considering it at the *(i+1)*-th step of the learning strategy.

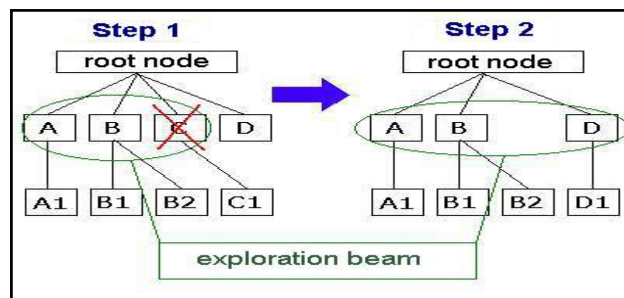


Fig.2. An example of search-tree pruning effect on beam-search width.

In order to maintain unchanged the width of the search beam, some grafting operations are necessary after pruning. Indeed, by removing the clauses that will be no more examined, the exploration beam decreases. Grafting operations aim to consider previous unspecialized clauses in order to restore the beam width, as shown in Fig.2.

Grafting operations are also necessary to preserve the generation of recursive clauses. For instance, by looking at the two specialization hierarchies of the predicate *odd* in Fig. 1, it is clear that once the clause *even(X)* ← *zero(X)* has been added to the empty theory (step 1), the consistent clause *odd(X)* ← *succ(Y,X), even(Y)* can be a proper node of the specialization hierarchy, since a base clause for the recursive definition of the predicate *even* is already available. Therefore, the grafting operations also aim to complete the pruned specialization hierarchy with new clauses that take predicate dependencies into account.

3.2 Caching for clause evaluation

Evaluating a clause corresponds to determining the lists of positive and negative examples covered by the clause itself. This requires a number of generalized implication tests, one for each positive or negative example. In ATRE the generalized implication test is optimized, however, if the number of tests to perform is high, the clause evaluation leads to efficiency problems anyway. To reduce the number of tests, we propose to cache the list of positive and negative examples of each clause, as well.

To clarify this caching technique, we distinguish between *dependent* clauses, that is, clauses with at least one literal in the body whose predicate symbol is a target predicate p_i, and independent clauses (all the others).

In independent clauses, the lists of negative examples remain unchanged between two subsequent learning steps. Indeed, the addition of a clause C to a partially learned theory T_i does not change the set of consequences of an independent clause, whose set of negative examples can neither increase nor decrease. Therefore, by caching the list of negative examples, the learning system can prevent its computation.

A different observation concerns the list of positive examples to be covered by the partially learned theory. For the same reason reported above it cannot increase, while it can decrease since some of the positive examples might have been covered by the added clause C. Actually, the set of positive examples of a clause C' generated at the $(i+1)$-th step can be calculated as intersection of the cached set computed at the i-th step of the learning strategy and the set of positive examples covered by the parent clause of C' in the specialization hierarchy computed at the $(i+1)$-th step (see Fig. 3). In the case of dependent clauses, both lists of the positive and negative examples can increase, decrease or remain unchanged, since the addition of a clause C to a partially learned theory T_i might change the set of consequences of a dependent clause. Therefore, caching the set of positive/negative examples covered by a dependent clause is useless.

It is noteworthy that, differently from the caching technique for clause generation, caching for clause evaluation does not require additional memory resources since all requested information are kept from the current learning step.

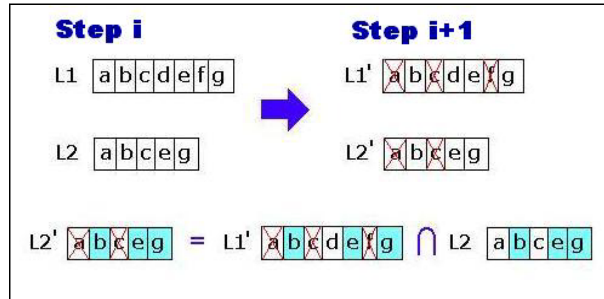


Fig.3. The positive examples list is calculated as intersection of the positive examples list of the same clause in previous learning step (i) and the positive examples list of the parent clause in current learning step (i+1).

4. Application to document understanding

The current release of ATRE is implemented in Sictus Prolog and C. It has also been integrated in WISDOM++(http://www.di.uniba.it/~malerba/wisdom++), an intelligent document processing system, that uses logical theories learned by ATRE to perform automatic classification and understanding of document images. In this section we show some experimental results for the document image understanding task alone.

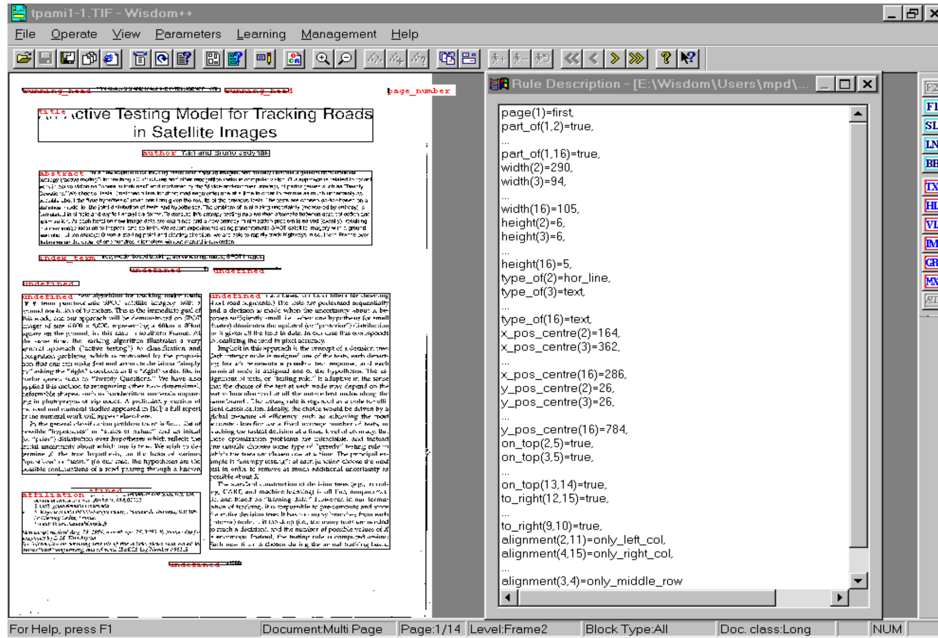


Fig. 4. Layout of a document image produced by WISDOM++ (left) and its partial description in the logical representation language adopted by ATRE (right).

A document is characterized by two different structures representing both its internal organization and its content: the layout structure and the logical structure. The former associates the content of a document with a hierarchy of layout components, while the latter associates the content of a document with a hierarchy of logical components. Here, the term document understanding denotes the process of mapping the layout structure of a document into the corresponding logical structure. The document understanding process is based on the assumption that documents can be understood by means of their layout structures alone. The mapping of the layout structure into the logical structure can be performed by means of a set of rules which can be generate automatically by learning from a set of training objects. Each training object describes the layout of a document image and the logical components associated to layout components (see Fig. 4).

To empirically investigate the effect of the proposed caching strategies, we selected twenty-one papers, published as either regular or short, in the IEEE Transactions on Pattern Analysis and Machine Intelligence, in the January and February issues of 1996. Each paper is a multi-page document; therefore, the dataset is composed by 197 document images in all. Since in the particular application domain, it generally happens that the presence of some logical components depends on the order page (e.g. *author* is in the first page), we have decomposed the document understanding problem into three learning subtasks, one for the first page of scientific papers, another for intermediate pages and the third for the last page. Target predicates are only unary and concern the following logical components of a typical scientific paper published in a journal: *abstract*, *affiliation*, *author*, *biography*, *caption*, *figure*, *formulae*, *index_term*, *page_number*, *references*, *running_head*, *table*, *title.* Some statistics on the dataset obtained from first page documents are reported in Table1.

Logical components	***Number of positive examples***
Abstract	21
Affiliation	22
Author	25
Index_term	11
Page_number	180
Running_head	203
Title	23
Total	**485**

Table 1. Distribution of logical components on first page documents.

By running ATRE on a document understanding dataset obtained from scientific papers, a set of theories is learned. Some examples of learned clauses follow:

```
author(X1) ← alignment(X1,X2)=only_middle_col, abstract(X2),
             height(X1)∈[7..13]
figure(X1) ← type_of(X1)= image, width(X1)∈[12..227],
             x_pos_centre(X1)∈[335..570]
references(X1) ← to_right(X1,X2), biografy(X2),
```

width(X2)∈[261..265]

They can be easily interpreted. For instance, the first clause states that if a quite short layout component (X1), whose height is between 7 and 13, is centrally aligned with another layout component (X2) labelled as the abstract of the scientific paper, then it can be classified as the author of the paper. These clauses show that ATRE can automatically discover meaningful dependencies between target predicates.

In the experiments the effect of the caching is investigated with respect to two system parameters, the minimum number of *consistent* clauses found at each learning step before selecting the best one and the beam of the search (*max_star* parameter). The former affects the depth of specialization hierarchies, in the sense that the higher the number of consistent clauses, the deeper the hierarchies. The latter affects the width of the search-tree.

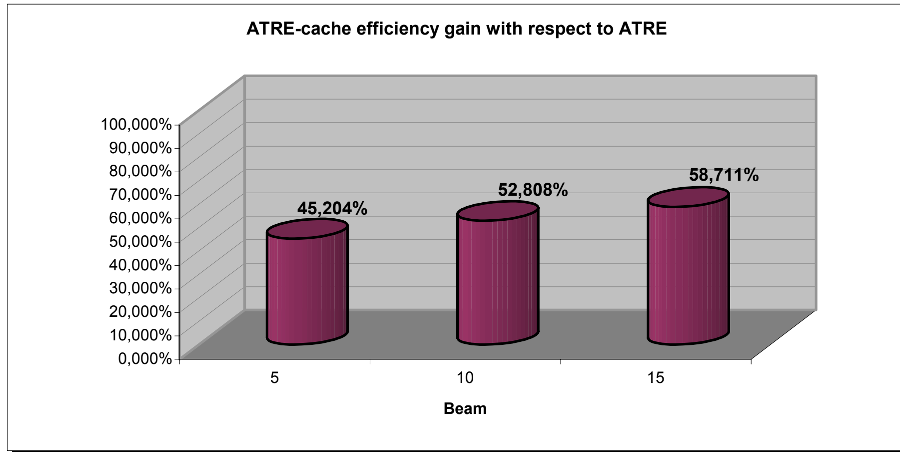


Fig. 5 Efficiency gain on "First-page" task on *max_star* variation.

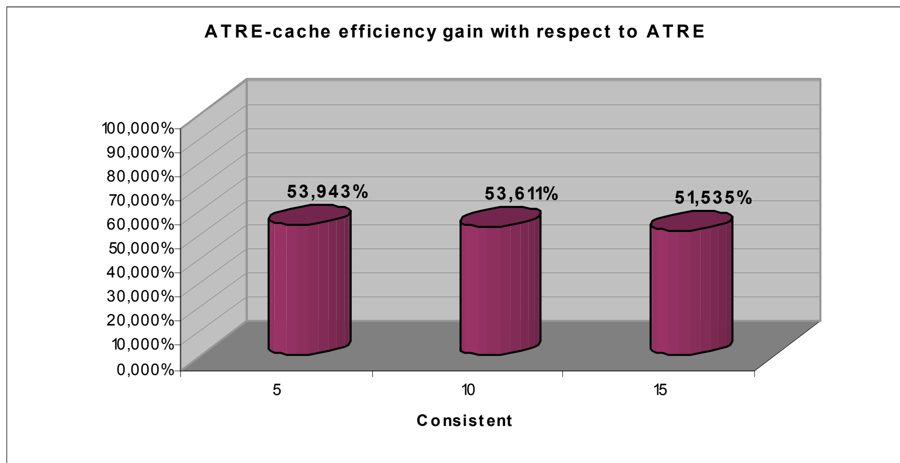


Fig. 6. Efficiency gain on "First-page" task on *consistent* variation.

Results for the first page learning task are shown in Figures 5 and 6. Percentages refer to reduction of the learning time required by ATRE with caching with respect to the original release of the system (without caching). Results show a positive dependence between the size of the beam and the reduction of the learning time. On the contrary, slight increases in the number of consistent clauses do not seem to significantly affect the efficiency gain due to caching.

5. Conclusions

In this paper issues and solutions of recursive theory learning are illustrated. Evolutions on the proposed search strategy to tackle efficiency problems are proposed. They have been implemented in ATRE and tested in the document understanding domain. Initial experimental results show that the learning task benefits from the caching strategy. As future work we plan to perform more extensive experiments to investigate the real efficiency gain in other real-world domains.

References

1. Apt, K.R.: Logic programming. In: van Leeuwen, J. (ed.): Handbook of Theoretical Computer Science, Vol. B. Elsevier, Amsterdam (1990) 493-574.
2. Boström, H.: Induction of Recursive Transfer Rules. In J. Cussens (ed.), Proceedings of the Language Logic and Learning Workshop (1999) 52-62.
3. Buntine, W.: Generalised subsumption and its applications to induction and redundancy. Artificial Intelligence, Vol. 36 (1988) 149-176.
4. Ceri, S., Gottlob, G., Tanca, L.: What you always wanted to know about Datalog (and never dared to ask). IEEE Trans. on Knowledge & Data Engineering 1(1) (1989) 146-166.
5. De Raedt, L., Dehaspe, L.: Clausal discovery. Machine Learning Journal, 26(2/3) (1997) 99-146.
6. De Raedt, L., Lavrac, N.: Multiple predicate learning in two Inductive Logic Programming settings. Journal on Pure and Applied Logic, 4(2) (1996) 227-254.
7. Khardon, R.: Learning to take Actions. Machine Learning, 35(1) (1999) 57-90.
8. Malerba, D.: Learning Recursive Theories in the Normal ILP Setting, Fundamenta Informaticae, 57(1) (2003) 39-77.
9. Mitchell, T.M.: Machine Learning. McGraw-Hill (1997).
10. Muggleton, S., Bryant, C.H.: Theory completion using inverse entailment. In: J. Cussens and A. Frisch (eds.): Inductive Logic Programming, Proceedings of the 10th International Conference ILP 2000, LNAI 1866, Springer, Berlin, Germany (2000) 130-146.
11. Nienhuys-Cheng, S.-W., de Wolf, R.: A complete method for program specialization based upon unfolding. Proc. 12th Europ. Conf. on Artificial Intelligence (1996) 438-442.
12. Nienhuys-Cheng, S.-W., de Wolf, R.: The Subsumption theorem in inductive logic programming: Facts and fallacies. In: De Raedt, L. (ed.): Advances in Inductive Logic Programming. IOS Press, Amsterdam (1996) 265-276.
13. Plotkin, G.D.: A note on inductive generalization. In: Meltzer, B., Michie, D. (eds.): Machine Intelligence 5. Edinburgh University Press, Edinburgh (1970) 153-163.
14. Plotkin, G.D.: A further note on inductive generalization. In: Meltzer, B., Michie, D. (eds.): Machine Intelligence 6. Edinburgh University Press, Edinburgh (1971) 101-124.
15. Rouveirol, C.: Flattening and saturation: Two representation changes for generalization. Machine Learning Journal, 14(2) (1994) 219-232.

From logic programs updates to action description updates *

J. J. Alferes[1], F. Banti[1], and A. Brogi[2]

[1] CENTRIA, Universidade Nova de Lisboa, Portugal,
jja|banti@di.fct.unl.pt
[2] Dipartimento di Informatica, Università di Pisa, Italy,
brogi@di.unipi.it

Abstract. An important branch of investigation in the field agents has been the definition of high level languages for representing effects of actions, the programs written in such languages being usually called action programs. Logic programming is an important area in the field of knowledge representation and some languages for specifying updates of Logic Programs had been defined. In this work we address the problem of establishing relationships between action programs and Logic Programming updates languages, particulary the newly defined Evolp language. Our investigation leads to the definition of a new paradigm for representing actions called Evolp action programs. We provide translations of some of the most known action description languages into Evolp action programs, and underline some peculiar features of this newly defined paradigm. One of such feature is that Evolp action programs can easily express changes in the very rules of the domains, including rules describing changes.

1 Introduction

In the last years the concept of agent had became central in the field of Artificial Intelligence. "*An agent is just something that acts*" [25]. Given the importance of the concept, ways of representing actions and their effects on the environment had been studied. A branch of investigation in this topic has been the definition of high level languages for representing effects of actions [7, 12, 14, 15], the programs written in such languages being usually called *action programs*. Action programs specify which facts (or fluents) change in the environment after the execution of a set of actions. Several works exist on the relation between these action languages and Logic Programming (LP) (e.g. [5, 12, 20]). However, despite the fact that LP has been successfully used as a language for declaratively representing knowledge, the mentioned works basically use LP for providing an operational semantics, and implementation, for action programs. This is so because normal logic programs [11], and most of their extensions, have no in-built

* This work was partially supported by project FLUX (POSI/40958/SRI/2001) and by project SOCS (IST-2001-32530).

means for dealing with changes, something which is quite fundamental for the relation with action languages.

In recent years some effort was devoted to explore the problem of how to update logic programs with new rules [3, 8, 9, 18, 19]. Here, knowledge is conveyed by sequences of programs, where each program in a sequence is an update to the previous ones. For determining the meaning of sequences of logic programs, rules from previous programs are assumed to hold by inertia after the updates (given by subsequent programs) unless rejected by some later rule. LP update languages [2, 4, 10, 18], besides giving meaning to sequences of logic programs, also provide in-built mechanisms for constructing such sequences. In other words, LP update languages extend LP by providing means to specify and reason about rule updates. In [5] the authors show, by examples, a possible use the LP update language LUPS [4] for representing actions. However, the work done does not establish an exact relationship between existing action languages and LP update languages and also the eventual advantages of LP update languages approach to actions are still not clear. The present work tries to clarify these points. Our investigation focuses on the newly defined Evolp language [2].

In section 2 we review some background and notation. In section 3 we show how to use macros defined in Evolp as an action description paradigm. Programs written in such macro language are called Evolp action programs (EAPs). We illustrate the usage of EAPs by an example involving a variant of the classical Yale Shooting Problem. In section 4 we establish the relationship between EAPs and existing approaches by providing simple translations of the action languages $\mathcal{A}$ [12], $\mathcal{B}$ [13] (which is a subset of the language proposed in [14]), and (the definite fragment of) $\mathcal{C}$ [15] into EAPs, thus showing that EAPs are *at least as expressive* as the cited action languages. Coming to this point the next question is what are the possible advantages of EAPs. The underlying idea of action frameworks is to describe dynamic environment. This is usually done by describing rules that specify, given a set of external actions, how the environment evolves. In a dynamic environment, however, not only the facts but also the "rules of the game" can change, in particular *the rules describing the changes*. The capability of describing such kind of *meta level changes* is, in hour opinion, an important feature of an action description language. In section 5 we address this topic in the context of EAPs and show EAPs seem, in this sense, more flexible than other paradigms. Evolp provides specific commands that allow for the specification of updates to the initial program but also provides the possibility to specify updates of these updates commands. We show, by successive elaborations of the Yale shooting example defined in section 3.1, how to use this feature to describe successive elaborations of the problem during the evolution of the environment. Finally, in section 6, we conclude and trace a route for future developments.

2 Background and notation

In this section we briefly recall syntax and semantics of *dynamic logic programs* [1] and the syntax and semantics for Evolp[2]. We also recall some basic notions and notation for action description languages.

2.1 Dynamic logic programs and Evolp

The main idea of logic programs updates is to update a logic program by another logic program or by a *sequence* of logic programs, also called *dynamic logic programs* (DLP) the initial program corresponding to the initial knowledge of a given (dynamic) domain, and the subsequent ones to successive updates of the domain. To represent negative information in logic programs and their updates, DLP requires generalized logic programs (GLPs) [21], which allows for default negation *not* A not only in the premises of rules but also in their heads. A language $\mathcal{L}$ is any set of propositional atoms. A literal in $\mathcal{L}$ is either an atom of $\mathcal{L}$ or the negation or such an atom. In general, given any set of atoms $\mathcal{F}$ we denote the by $\mathcal{F}_{Lit}$ the set of literals over $\mathcal{F}$. Given a literal L, if $L = Q$, where Q is an atom, by *not* L we denote the negative literal *not* Q. Viceversa, if $L = not\ Q$, by *not* L we denote the atom Q. A GLP defined over a propositional language $\mathcal{L}$ is a set of rules of the form $L \leftarrow Body$, where L is a literal in $\mathcal{L}$, and $Body$ is a *set* of literals in $\mathcal{L}$.[3] We say a set of literals $Body$ is true in an interpretation I (or that I satisfies $Body$) iff $Body \subseteq I$. In the paper we will use programs containing variables. As usual when programming within the stable models semantics, a program with variables stands for the propositional program obtained as the set of all possible ground instantiation of the program.

Two rules τ and η are *conflicting* (denoted by $\tau \bowtie \eta$) iff the head of τ is the atom A and the head of η is *not* A or viceversa. A dynamic logic program over a language $\mathcal{L}$ is a sequence $P_1 \oplus \ldots \oplus P_m$ (also denoted $\oplus P_i^m$) where the P_is are GLPs defined over $\mathcal{L}$. The *refined stable model semantics* of DLP defined in [1] assigns to each sequence $P_1 \oplus \ldots \oplus P_n$ a set of stable models (that is proven there to coincide with the stable models based semantics defined in [21] when the sequence is formed by a single GLP). The rationale for the definition of a stable model M of a DLP is made in accordance with the **causal rejection principle** [9, 18]: If the body of a rule in a given update is true in M the considered rule rejects all the conflicting rules in previous updates, which means that such rules are ignored in the computation of the stable model. In the refined semantics for DLPs such rule also rejects any conflicting rule in the same update. Moreover, an atom A is assumed false by default if there is no rule, in none of the programs in the sequence, with head A and a true body in M. Formally:

$$Default(\oplus P_i^m, M) = \{not\ A \mid \ \nexists\ A \leftarrow Body \in \bigcup P_i \quad \wedge Body \subseteq M\}$$

$$Rej^S(\oplus P_i^m, M) = \{\tau \mid \tau \in P_i :\ \exists\ \eta \in P_j\ i \leq j,\ \tau \bowtie \eta\ \wedge\ Body(\eta) \subseteq M\}$$

where M is an interpretation, i.e. any set of literals in $\mathcal{L}$ such that, for each atom A, either $A \in\ M$ or *not* $A \in\ M$. If $\oplus P_i^m$ is clear from the context, we omit it as first argument of the above functions.

[3] Note that, by defining rule bodies as sets, the order and number of occurrences of literals does not matter.

Definition 1. *Let $\oplus P_i^m$ be a DLP over language $\mathcal{L}$ and M a interpretation. M is a refined stable model of $\oplus P_i^m$ iff*

$$M = least\left(\bigcup P_i \setminus Rej^S(M)) \cup Default(M)\right)$$

where $least(P)$ denotes the least Herbrand model of the definite program [22] obtained by considering each negative literal not A in P as a new atom.

Having defined the meaning of sequences of programs, we are left with the problem of how to come up with those sequences. This is the subject of LP update languages [2, 4, 10, 18]. Among the existing languages, Evolp [2] uses a particulary simple syntax, which extends the usual syntax of GLPs by introducing the special predicate $assert/1$. Given any language $\mathcal{L}$, the language $\mathcal{L}_{assert}$ is recursively defined as follows: every atom in $\mathcal{L}$ is also in $\mathcal{L}_{assert}$; for any rule τ over $\mathcal{L}_{assert}$, the atom $assert(\tau)$ is in $\mathcal{L}_{assert}$; nothing else is in $\mathcal{L}_{assert}$. An *Evolp program* over $\mathcal{L}$ is any GLP over $\mathcal{L}_{assert}$. An *Evolp sequence* is a sequence (or DLP) of Evolp programs. The rules of an Evolp program are called *Evolp rules*.

Intuitively an expression $assert(\tau)$ stands for "update the program with the rule τ". Notice the possibility in the language to nest an assert expression in another. The intuition behind the Evolp semantics is quite simple. Starting from the initial Evolp sequence $\oplus P_i^m$ we compute the set, $\mathcal{SM}(\oplus P_i^m)$, of the stable models of $\oplus P_i^m$. Then, for any element M in $\mathcal{SM}(\oplus P_i^m)$, we update the initial sequence with the program P_{m+1} consisting of the set of rules τ such that the atom $assert(\tau)$ belongs to M. In this way we obtain the sequence $\oplus P_i^m \oplus P_{m+1}$. Since $\mathcal{SM}(\oplus P_i^m)$ contains, in general, several models we may have different lines of evolution. The process continues by obtaining the various $\mathcal{SM}(\oplus P_i^{m+1})$ and, with them, various $\oplus P_i^{m+2}$. Intuitively, the program starts at step 1 already containing the sequence $\oplus P_i^m$. Then it updates itself with the rules asserted at step 1, thus obtaining step 2. Then, again, it updates itself with the rules asserted at this step, and so on. The evolution of any Evolp sequence can also be influenced by external events. An external event is itself an Evolp program. If, at a given step n, the programs receives the external update E_n, the rules in E_n are added to the last self update for the purpose of computing the stable models determining the next evolution but, in the successive step $n+1$ they are no longer considered (that's why they are called *events*). Formally:

Definition 2. *Let n be a natural number. An* evolution interpretation *of length n, of an evolving logic program $\oplus P_i^m$ with an event sequence $\oplus E_i$ is any finite sequence $\mathcal{M} = M_1, \ldots, M_n$ of interpretations over $\mathcal{L}_{assert}$. The* evolution trace *$Tr(P)$ associated with an evolution interpretation $M_1, \ldots, M_n$ is the sequence $P_1 \oplus \ldots \oplus P_{n+m}$ where $P_{m+i} = \{\tau \mid assert(\tau) \in M_{i-1}\}$ for $m+1 < i \leq n+m$*

Definition 3. *Let $\oplus P_i^m$ be any Evolp sequence with external events $\oplus E_i^n$ (where n is a natural number), and $\mathcal{M} = M_1, \ldots, M_n$ be an evolving interpretation of length n with trace $P_1 \oplus \ldots \oplus P_{n+m}$. $\mathcal{M}$ is an evolving stable model of $\oplus P_i^m$ with event sequence $\oplus E_i$ at step n iff M_k is a refined stable model of the program $P_1 \oplus \ldots \oplus (P_k \cup E_k)$ for any k with $m+1 \leq k \leq n+m$.*

2.2 Action languages

The purpose of an action language is to provide ways of describing how an environment evolves given a set of external actions. A specific environment that can be modified through external actions is called an *action domain*. To any action domain we associate a pair of sets of atoms $\mathcal{F}$ and $\mathcal{A}$. We call the elements of $\mathcal{F}$ *fluent atoms* or simply *fluents* and the elements of $\mathcal{A}$ *action atoms* or simply *actions*. Basically the fluents are the observable in the environment and the actions are, clearly, the external actions. A *fluent literal* (resp. *action literal*) is an element of $\mathcal{F}_{Lit}$ (resp. an element of $\mathcal{A}_{Lit}$). In the following, Q will be in general a fluent atom, F a fluent literal and A an action atom. A *state of the world* (or simply a *state*) is any interpretation over $\mathcal{F}$. We say a fluent literal F is true at a given state s iff F belongs to s.

Each action language provides ways of describing action domains through sets of expression called an *action programs*. Usually, the semantics of an action program is defined in terms of a *transition system* i.e. a function whose argument is any pair (s, K), where s is a state of the world and K is a subset of $\mathcal{A}$, and whose value is any set of states of the world. Intuitively, given the current state of the world, a transition system specifies which are the possible resulting states after performing, simultaneously, all the actions in K.

Two kinds of expressions that are common within action description languages are *static and dynamic rules*. The *static rules* basically describe the rule of the domain, while *dynamic rules* describe effects of actions. A dynamic rule has a set of *preconditions*, namely conditions that have to be satisfied in the present state in order to have a particular effect in the future state, and *post-conditions* describing such an effect.

In the following we will consider three existing action languages, namely: $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$. The language $\mathcal{A}$ [13] is very simple, allowing only dynamic rules of the form A **causes** F **if** $Cond$ where $Cond$ is a conjunction of fluent literals, such rule intuitively means: performing the action A causes L to be true in the next state if $Cond$ is true in the current state. The language $\mathcal{B}$ [13] is an extension of $\mathcal{A}$ which also considers static rules, i.e. expression of the form F **if** $Body$ where $Body$ is a conjunction of fluent literals which, intuitively, means: if $Body$ is true in the current state, then F is also true in the current state. A fundamental notion in both $\mathcal{A}$ and $\mathcal{B}$ is *fluent inertia* [13]. A fluent F is inertial if its truth value is preserved from a state to another, unless it is changed by the (direct or indirect) effect of an action. For a detailed definition of the semantics of $\mathcal{A}$ and $\mathcal{B}$ see [13].

Static and dynamic rules are also the bricks of the action language $\mathcal{C}$ [16, 15]. Static rules in $\mathcal{C}$ are of the form **caused** J **if** H while dynamic rules are of the form **caused** J **if** H **after** O where J and H are formulae such that any literal in them is a fluent literal and O is any formula such that any literal in it is a fluent or an action literal. The formula O is the precondition of the dynamic rule and the static rule **caused** J **if** H is its postcondition. The semantic of $\mathcal{C}$ is based on *causal theories*[15]. Casual theories are sets of rules of the form **caused** J **if** H meaning: If H is true this is an explanation for J. Within causal theories is that

something is true iff it is caused by something else. Given any action program P, a state s and a set of actions K, we consider the causal theory T given by the static rules of P and the postconditions of the dynamic rules whose preconditions are true in $s \cup K$. Then s' is a possible resulting state iff it is a casual model of T. For a more detailed background on action languages see [12].

3 Evolp action programs

As we have seen, Evolp and action description languages share the idea of a system that evolves. In both, the evolution is influenced by external events (respectively, updates and actions). Evolp is actually a programming language devised for representing any kind of computational problem, while action description languages are devised for the specific purpose of describing actions. A natural idea is then to develop special kind of Evolp sequences for representing actions and then compare such kind of programs with existing action description languages. We will call this kind of programs *Evolp Action Programs* (EAPs).

Following the philosophy of Evolp we use the basic construct *assert* for defining special-purpose macros. As it happens for other action description languages, EAPs are defined over a set of fluents $\mathcal{F}$ and a set of actions $\mathcal{A}$. A state of the world, in EAPs, is any interpretation over $\mathcal{F}$. To describe action domains we use an initial Evolp sequence, $I \oplus D$. The Evolp program D contains the description of the environment, while I contains some initial declarations, as it will be clarified later. As in $\mathcal{B}$ and $\mathcal{C}$, EAPs contain static and dynamic rules.

A *static rule* is simply an Evolp rule of the form $F \leftarrow Body$ where F is a fluent literal and $Body$ is a set of fluent literals.

A *dynamic rule* over $(\mathcal{F}, \mathcal{A})$ is a (macro) expression $\mathbf{effect}(\tau) \leftarrow Cond$ where τ is any static rule $F \leftarrow Body$ and $Cond$ is any set of fluent or action literals. Such an expression simply stand for the following set of Evolp rules:

$$F \leftarrow Body,\ event(L \leftarrow Body)\ (1) \qquad assert(event(F \leftarrow Body)) \leftarrow Cond.\ (2)$$
$$assert(not\ event(F \leftarrow Body)) \leftarrow event(\tau), not\ assert(event(F \leftarrow Body))\ (3)$$

where $event(F \leftarrow Body)$ is a new literal. The intuitive meaning of such a rule is that the static rule τ has to be considered *only* in those states whose predecessor satisfies condition $Cond$. Since some of the conditions literals in $Cond$ may be action atoms, such a rule may describe the effect of a given set of actions under some conditions. In fact, the above set of rules fits with this intuitive meaning. Rule (1) is not applicable whenever $event(L \leftarrow Body)$ is false. If at some step n the conditions $Cond$ are satisfied, then, by rule (2), $event(L \leftarrow Body)$ becomes true at step $n+1$. Hence, at step $n+1$, the rule (1) will play the same role as static rule $F \leftarrow Body$. If at step $n+1$ $Cond$ is no longer satisfied, then, by rule (3) the literal $event(L \leftarrow Body)$ will become false again and then the rule (1) will be again not effective. The behaviour of **effect** is different from the *assert* command. If we assert τ, it remains by inertia, while with **effect** it lasts for one step only. Moreover, if we assert τ, such rule could reject another rule while a rule inside an **effect** expression does not reject static rules.

Besides static and dynamic rules, we still need another brick to complete our construction. As we have seen in the description of the $\mathcal{B}$ language, a notable concept is fluent inertia. This idea is not explicit in Evolp where *the rules* (and not the fluents) are preserved by inertia. Nevertheless, we can show how to obtain fluent inertia using macro programming in Evolp. An *inertial declarations* over $(\mathcal{F}, \mathcal{A})$ is a (macro) expression **inertial**$(\mathcal{K})$, where $\mathcal{K} \subseteq F$. The intended intuitive meaning of such expression is that the fluents in $\mathcal{K}$ are inertial. Before defining what this expression stands for, we state that the program I is always of the form **initialize**$(\mathcal{F})$, where **initialize**$(\mathcal{F})$ stands for the set of rules (where F is any fluent literal in $\mathcal{F}_{Lit}$, and $prev(F)$ are new atoms not in $\mathcal{F} \cup \mathcal{A}$): $F \leftarrow prev(F)$ The *inertial declaration* **inertial**$(\mathcal{K})$ stands for the set (where F ranges over $\mathcal{K}$):

$$assert(prev(F)) \leftarrow F. \quad assert(not\ prev(F)) \leftarrow not\ F.$$

Let us consider the behaviour of this macro. If we do not declare F as an inertial fluent the rule $F \leftarrow prev(F)$ has no effect. If we declare F as an inertial literal, $prev(F)$ is true in the current state iff in the previous state F was true. Hence in this case F is true in the current state *unless* there is a static or dynamic rule that rejects such assumption. Viceversa, if F was false in the previous state then, F is true in the current one iff it is derived by a static or dynamic rule. We are now ready to formalize the syntax of Evolp action programs.

Definition 4. *Let $\mathcal{F}$ and $\mathcal{A}$ be two disjoint sets of propositional atoms. An Evolp action program (EAP) over $(\mathcal{F},\ \mathcal{A})$ is any Evolp sequence $I \oplus D$ where : $I = Initialize(\mathcal{F})$, and D is any set consisting of static rules, dynamic rules and inertial declarations over $(\mathcal{F}, \mathcal{A})$*

Given an Evolp action program $I \oplus D$, the initial state of the world s (which, as stated above is an interpretation over $\mathcal{F}$) is passed to the program together with the set K of the actions performed at s, as part of an external event. A resulting state is the last element of any evolving stable model of $I \oplus D$ given the event $s \cup K$ restricted to the set of fluent literals. I.e:

Definition 5. *Let $I \oplus D$ be any EAP over $(\mathcal{F},\ \mathcal{A})$ and s a state of the world. Then s' is a resulting state from s given $I \oplus D$ and the set of actions K iff there exists an evolving stable model M_1, M_2 of $I \oplus D$ given the external event $s \cup K$ such that $s' \equiv_{\mathcal{F}} M_2$(where by $s' \equiv_{\mathcal{F}} \mathcal{M}_2$ we simply mean $s' \cap \mathcal{F}_{Lit} = M_2 \cap \mathcal{F}_{Lit}$).*

The definition can be immediately generalized to sequences of set of actions.

Definition 6. *Let $I \oplus D$ be any EAP and s a state of the world. Then s' is a resulting state from s given $I \oplus D$ and the sequence of actions $K_1 \ldots,\ K_n$ iff there exists an evolving stable model $M_1, \ldots M_n$ of $I \oplus D$ given the external event $s \cup K_1, \ldots,\ K_n$ such that $s' \equiv_{\mathcal{F}} M_n$.*

Since EAPs are based on the Evolp semantics, which is an extension of the stable model semantics for normal logic programs, we can easily prove that the complexity of the computation of the two semantics is the same.

Theorem 1. *Let $I \oplus D$ be any EAP over $(\mathcal{F}, \mathcal{A})$, s a state of the world and $K \subseteq \mathcal{A}$. To find a resulting state s' from s given $I \oplus D$ and the set of actions K is an NP-hard problem.*

It is important to notice that, if the initial state s does not satisfies the static rules of the EAP, the correspondent Evolp sequence has no stable model, and hence there will be no successor state. From now onwards we assume that the initial state satisfies the static rules of the domain.

We now show an example of usage of EAPs by elaborating on probably the most famous example of reasoning about actions. The presented elaboration highlights some important features of EAPs: the possibility of handling non-deterministic effects of actions, non-inertial fluents, non-executable actions, and effects of actions lasting for just one state.

3.1 An elaboration of the Yale shooting problem

In the original Yale shooting problem [26], there is a single-shot gun which is initially unloaded, and a turkey which is initially alive. We can load the gun and shoot the turkey. If we shoot, the gun becomes unloaded and the turkey dies. We consider a slightly more complex scenario where there are several turkeys and where the shooting action refers to a specific turkey. Each time we *shoot* a specific turkey, we either hit and kill the bird or miss it. Moreover the gun becomes unloaded and there is be a bang. It is not possible to shoot with an unloaded gun. We also add the property that any turkey moves iff it is not dead.

For expressing the non executable the problem we make use of a standard technique used in LP under the stable model semantics. Suppose the used EAP contains dynamic rules of the form $\mathbf{effect(}u \leftarrow not\ u\mathbf{)} \leftarrow Cond$ where u is a literal which does not appear elsewhere. In the following we use, for such rules, the notation $\mathbf{effect(}\bot\mathbf{)} \leftarrow Cond$. This kind of rules means that, if $Cond$ is true in the current state, then there is no resulting state. This come from the known fact that programs containing $u \leftarrow not\ u$ has no stable models.

To represent this situation we use the set of fluents: $\{dead(X), moving(X), missed(X), hit(X),\ loaded,\ bang\}$ plus the auxiliary fluent u, and the actions $load$ and $shoot(X)$ (where the X is instantiated with the various turkeys). The fluents *dead* and *loaded* are inertial fluents, since their truth value should remain unchanged until modified by some action effect. The fluents *missed*, *hit* and *bang* are not inertial. Finally, for every turkey t, the fluent $moving(t)$ is not declared as inertial. The problem is encoded by the EAP $I \oplus D$, where $I =$ $\mathbf{initialize(}loaded,\ moving(X),\ dead\ missed(X),\ hit(X), u\mathbf{)}$, and D is

$$
\begin{array}{ll}
\mathbf{effect(}loaded\mathbf{)} \leftarrow load. & moving(X) \leftarrow not\ dead(X) \\
\mathbf{effect(}\bot\mathbf{)} \leftarrow shoot(X),\ not\ loaded & \mathbf{effect(}not\ loaded.\mathbf{)} \leftarrow shoot(X) \\
\mathbf{effect(}dead(X) \leftarrow hit(X)\mathbf{)} \leftarrow\ shoot(X) & \mathbf{effect(}bang\mathbf{)} \leftarrow shoot(X) \\
\mathbf{effect(}hit(X) \leftarrow not\ missed(X)\mathbf{)} \leftarrow shoot(X) & \mathbf{inertial(}loaded\mathbf{)} \\
\mathbf{effect(}missed(X) \leftarrow not\ hit(X)\mathbf{)} \leftarrow shoot(X) & \mathbf{inertial(}dead(X)\mathbf{)}
\end{array}
$$

Let us analyze this EAP. Rule $\mathbf{effect(}\bot\mathbf{)} \leftarrow shoot(X),\ not\ loaded$ encodes the impossibility to execute the action $shoot(X)$ when the gun is unloaded.

The static rule $moving(X) \leftarrow not\ dead(X)$ implies that, for any turkey tk, $moving(tk)$ is true if $dead(tk)$ is false. Since this is the unique rule for $moving(tk)$ we obtain that $moving(tk)$ is true iff $dead(tk)$ is true. Notice that declaring $moving(tk)$ as inertial, would result, in our description, in the possibility of having a moving dead turkey! In fact, suppose we insert **inertial(**$moving(X)$**)** in the EAP above. Suppose further that $moving(tk)$ is true at state s, that we shoot at tk and kill it. Since $moving(tk)$ is an inertial fluent, in the resulting state $dead(tk)$ is true but $moving(tk)$ also remains true by inertia. Also notable is how effects that last only for one state, like the noise provoked by the shoot are easily encoded. The last three dynamic rules encodes a non deterministic behaviour, each shoot action can either hit and kill a turkey or miss it.

We provide an example of a possible evolution. In the following we adopt the usual convention of the Stable Models semantics where we omit the negative literals belonging to an interpretation, hence any interpretation is represented as a set of atoms. Let us consider the initial state $\{\}$. The state will remain unchanged until we perform some action. If we load the the gun, the program is updated by the external event $\{load\}$. In the unique successor state, the fluent *loaded* is true and nothing else is changed. The truth value of the fluent remains unchanged (by inertia) until we perform some other action. The same applies for the fluents $dead(t)$ where tk is any turkey. The fluents $bang, missed(tk), hit(tk)$ remains false by default. If we shoot at a specific turkey (let us call the turkey Smith) we update the program with the event $shoot(smith)$. Now several things happen. First, *loaded* become $false$, and *bang* becomes true, as an effect of the action. Moreover, the rules $hit(smith) \leftarrow missed(smith)$ and $missed(smith) \leftarrow hit(smith)$ are considered as rules of the domain for one state. As a consequence we can have two possible resulting states. In the first one $missed(smith)$ is true, and all the others fluents are false. In the second one $hit(smith)$ is true, $missed(smith)$ is false and, by the static rule $dead(X) \leftarrow hit(X)$, the fluent $dead(smith)$ becomes true. In both the resulting states, nothing happens to the truth value of $dead(tk),\ hit(tk)$ and $dead(tk)$ for $tk \neq smith$.

4 Relationship to existing action languages

In this section we show embeddings into EAPs of the action languages $\mathcal{B}$ and (the definite fragment of) $\mathcal{C}$ [4]. We will assume that the considered initial states are consistent wrt the static rules of the program, i.e. if the body of a static rule is true in the considered state, the head is true as well.

Let us consider first the $\mathcal{B}$ language. The basic ideas of static and dynamic rules of $\mathcal{B}$ and EAPs are very similar. The main difference between the two is that in $\mathcal{B}$ *all* the fluents are considered as inertial, whilst in EAPs only those that are declared as such are inertial. The translation of $\mathcal{B}$ into EAPs is then straightforward: All fluents are declared as inertial and then the syntax of static

[4] The embedding of language $\mathcal{A}$ is not explicitly exposed here since $\mathcal{A}$ is a (proper) subset of the $\mathcal{B}$ language.

and dynamic rules is adapted. In the following we use, with abuse of notation, *Body* and *Cond* both for conjunctions of literals and for sets of literals.

Definition 7. *Let P be any action program in $\mathcal{B}$ over the fluent language $\mathcal{F}$. The translation $B(P, \mathcal{F})$ is the couple $(I^B \oplus D^{BP}, \mathcal{F}^B)$ where: $\mathcal{F}^B \equiv \mathcal{F}$, $I^B =$* ***initialize($\mathcal{F}$)*** *and D^{BP} contains exactly the following rules:*

- ***inertial(F)*** *for each fluent $F \in \mathcal{F}$*
- *a rule $L \leftarrow Body$ for any static rule L* ***if*** *$Body$ in P.*
- *a rule* ***effect(L)*** *$\leftarrow A,\ Cond.$ for any dynamic rule A* ***causes*** *L* ***if*** *$Cond$ in P.*

Theorem 2. *Let P be any $\mathcal{B}$ action program over $\mathcal{F}$, $(I^B \oplus D^{BP}, \mathcal{F})$ its translation, s a state and K any set of actions. Then s' is a resulting state from s given P and the set of actions K iff it is a resulting state from s given $I^B \oplus D^{BP}$ and the set of actions K.*

Let us consider now the action language $\mathcal{C}$. It is known that the computation of the possible resulting states in the full $\mathcal{C}$ language is $\sum_P^2$-hard, [15]. So, this language belongs to a category with higher complexity than EAPs which are NP-hard. However, only a fragment of $\mathcal{C}$ is implemented and the complexity of such fragment is NP. This fragment is known as the *definite fragment* of $\mathcal{C}$ [15]. In such fragment static rules are expressions of the form **caused** F **if** *Body* where F is a fluent literal and *Body* is a conjunction of fluent literals, while dynamic rules are expressions of the form **caused** *not* F **if** *Body* **after** *Cond* where *Cond* is a conjunction of fluent or action literals[5]. For this fragment it is possible to provide a translation into EAPs.

The main problem of the translation of $\mathcal{C}$ into EAPs lies the simulation of causal reasoning with stable model semantics. The approach followed here to encode causal reasoning with stable models is in line with the one proposed in [20]. We need to introduce some auxiliary predicates and define a syntactic transformation of rules. Let $\mathcal{F}$ be a set of fluents, by $\mathcal{F}^C$ we denote the set of fluents $\mathcal{F} \cup \{F_N \mid F \in \mathcal{F}\}$. We add, for each $F \in \mathcal{F}$, the constraints:

$$\leftarrow not\ F,\ not\ F_N. \quad \leftarrow F,\ F_N. \quad (2)$$

Let F be a fluent and $Body = F_1, \ldots, F_n$ a conjunction of fluent literals. We will use the following notation: $\overline{F} = not\ F_N$, $\overline{not\ F} = not\ F$ and $\overline{Body} = \overline{F_1}, \ldots, \overline{F_n}$

Definition 8. *Let P be any action program in $\mathcal{C}$ over the fluent language $\mathcal{F}$. The translation $C(P, \mathcal{F})$ is the couple $(I^C \oplus D^{CP}, \mathcal{F}^C)$ where: $\mathcal{F}^C$ is defined as above, $I^C \equiv$* ***initialize($\mathcal{F}^C$)*** *and D^{CP} consists exactly of the following rules:*

- *a rule* ***effect($F \leftarrow \overline{Body}$)*** *$\leftarrow Cond$, for any dynamic rule in P of the form* ***caused*** *F* ***if*** *$Body$* ***after*** *$Cond$;*

[5] The definite fragment defined in [15] is (apparently) more general, allowing *Cond* and *Body* to be arbitrary formulae. However, it is easy to prove that such kind of expressions are equivalent to a set of expressions of the form described above

- *a rule* $\textbf{\textit{effect(}}F_N \leftarrow \overline{Body}\textbf{\textit{)}} \leftarrow Cond$*, for any dynamic rule in* P *of the form* ***caused*** *not* F ***if*** *Body* ***after*** *Cond;*
- *a rule* $F \leftarrow \overline{Body}$*, for any static rule in* P *of the form* ***caused*** F ***if*** *Body;*
- *a rule* $F_N \leftarrow \overline{Body}$*, for any static rule in* P *of the form* ***caused*** *not* F ***if*** *Body;*
- *The rules (2) for each fluent in* $\mathcal{F}$*.*

For this translation we obtain a result similar to the one obtained for the translations of the $\mathcal{B}$ language. In this case:

Theorem 3. *Let* P *be any* $\mathcal{C}$ *action program over* $\mathcal{F}$*,* $(I^C \oplus D^{CP}, \mathcal{F}^C)$ *its translation,* s *a state,* s^C *the interpretation over* $\mathcal{F}^C$ *defined as follows:*

$$s^C = s \cup \{\overline{Q} \mid Q \in s\} \cup \{not\ \overline{Q} \mid not\ Q \in s\}$$

and K *any set of actions. Then* s^* *is a resulting state from* s^C *given* $I^C \oplus D^{CP}$ *and the set of actions* K *iff there exists* s' *such that* s' *is a resulting state from* s*, given* P *and the set of actions* K*.*

By showing translations of the action languages $\mathcal{B}$ and $\mathcal{C}$ into EAPs, we proved that EAPs are *at least as expressive* as such languages. Moreover the provided translations are quite simple (basically one EAP static or dynamic rule for each static or dynamic rule in the other languages). The next natural question is: Are they *more expressive*?

5 Updates of action domains

Action description languages describe the rules governing a domain where actions are performed. In practical situations, it may happen that the very rules of the domain change with time too. EAPs are just a particular kind of Evolp sequences. So, as in general Evolp sequences they can be updated by external events.

When we want to update the existing rules by the rule τ, we just add the fact $assert(\tau)$ as an external event. This way, the rule τ is asserted and the existing Evolp sequence is updated. Following this line, we extend EAPs, allowing the external events updating an EAP to contain facts of the form $assert(\tau)$ where τ is an Evolp rule and show how they can be used to express updates to EAPs.

To illustrate how to update an EAP, we come back to the example of section 3.1. Let $I \oplus D$ be the EAP defined in that section. Let us now consider that after some shots, and dead turkeys, we acquire rubber bullets. We can now either load the gun with normal bullets or with a rubber bullet, but not with both. If we shoot with a rubber loaded gun, we never kill a turkey.

To describe this change in the domain, we introduce a new inertial fluent representing the gun being loaded with rubber bullets. We have to express that, if the gun is rubber-loaded, we can not kill the turkey. For this purpose we introduce the new macro: *not* $\textbf{effect}(F \leftarrow Body) \leftarrow Cond$ where F, is a fluent literal, *Body* is a set of fluents literals and *Cond* is a set of fluent or action literals. We refer to such expressions as *effects inhibitions*. This macro simply stands for the rule $assert(not\ event(F \leftarrow Body)) \leftarrow Cond$ where $event(F \leftarrow Body)$ is a

new atom. The intuitive meaning is that, if the condition $Cond$ is true in the current state, any dynamic rule whose effect is the rule $F \leftarrow Body$ is ignored.

To encode the changes described above, we update the EAP with the external event E_1 consisting of the facts $assert(I_1)$ where $I_1 = (\mathbf{initialize(}rubber_loaded\mathbf{)})$. Then, in the subsequent state, we update the program with the external update $E_2 = assert(D_1)$ where D_1 is the set of rules[6]

$$
\begin{array}{l}
\mathbf{effect(}\bot\mathbf{)} \leftarrow rubber_loaded,\ load. \\
\mathbf{inertial(}rubber_loaded\mathbf{)} \\
\mathbf{effect(}\bot\mathbf{)} \leftarrow loaded,\ rubber_load. \\
not\ \mathbf{effect(}dead(X) \leftarrow hit(X)\mathbf{)} \leftarrow rubber_loaded.
\end{array}
$$

Let us analyze the proposed updates. First, the fluent $rubber_loaded$ is initialized. It is important to initialize any fluent before starting to use it. The newly introduced fluent is declared as inertial and two dynamic rules are added specifying that load actions are not executable when the gun is already loaded in a different way. Finally we use the new command to specify that the effect $dead(X) \leftarrow hit(X)$ does not occurs if, in the previous state, the gun was loaded with rubber bullets. Since this update is more recent than the original rule $\mathbf{effect(}dead(X) \leftarrow hit(X)\mathbf{)} \leftarrow shoot(X)$, such dynamic rule is updated.

It is also possible to update static rules and the descriptions of effects of an action. Suppose the cylinder of the gun becomes dirty and, whenever one shoots, the gun may either work properly or fail. If the gun fails, the action *shoot* has no effect. We introduce two new fluents in the program with the event $assert(I_2)$ where $I_2 = \mathbf{initialize(}fails, work\mathbf{)})$. Then, we assert the event $E_2 = assert(D_2)$ where D_2 is the following EAP

$$
\begin{array}{ll}
\mathbf{effect(}fails \leftarrow not\ work\mathbf{)} \leftarrow shoot(X). & \mathbf{effect(}work \leftarrow not\ fails\mathbf{)} \leftarrow shoot(X). \\
not\ bang \leftarrow fails. & not\ unloaded \leftarrow fails. \\
not\ missed \leftarrow fails. & not\ missed \leftarrow fails.
\end{array}
$$

This last example is important since it shows how to update the effects of a dynamic rule via a new static rule. It is also possible to update the effects of a dynamic rule via another dynamic rule. We show a possible evolution of the updated system. Suppose currently the gun is not loaded. We load the gun with a rubber bullet and then we shoot to the turkey named Trevor. The initial state is $\{\}$. The first set of actions is $\{rubber_load\}$ The resulting state after this action is $s' \equiv \{rubber_loaded\}$. Suppose we perform the action *load*. Since the EAP is updated with the dynamic rule $\mathbf{effect(}\bot\mathbf{)} \leftarrow rubber_loaded,\ load.$ there is no resulting state. This happens because we have performed a non executable action. Suppose, instead, the second set of actions is $\{shoot(trevor)\}$. There are three possible resulting states. In one the gun fails. In this case, the resulting state is, again, s'. In the second, the gun works but the bullet misses Trevor. In this case, the resulting state is $s''_1 \equiv \{missed(trevor)\}$. Finally, the gun works

[6] In the remainder we use the notation $assert(U)$, where U is a set of macros (which are themselves sets of Evolp rules) for the set of all facts $assert(\tau)$ such that τ is a rule used in: there exists a macro η in U with $\tau\ \in \eta$.

and the bullet hits Trevor. Since the bullet is a rubber bullet, Trevor is still alive. In this case the resulting state is $s''_2 \equiv \{hit(trevor)\}$.

The events introduced changes in the behaviour of the original EAP. This opens a new problem. In classical action languages we do not care about the previous *history* of the world: If the current state of the world is s, the computation of the resulting states is not affected by the states before s. In the case of EAPs the situation is different, since external updates can change the behaviour of the considered EAP. Fortunately, we do not have to care about the *whole* history of the world, but just about those events containing new initializations, inertial declarations, effects inhibitions, and static and dynamic rules.

It is possible to have a compact description of an EAP that is updated several times via external events. For that we need to further extend the original definition of EAPs.

Definition 9. *An* updated *Evolp action program over* $(\mathcal{F}, \mathcal{A})$ *is any sequence* $I \oplus D_1 \oplus \ldots \oplus D_n$ *where* I *is* ***initialize(***$\mathcal{F}$***)***, and the various* D_k *are sets consisting of static rules, dynamic rules, inertial declarations and effects inhibitions such that any fluent appearing in* D_k *belongs to* $\mathcal{F}$

In general, if we update an Evolp action program $I \oplus D$ with the subsequent events $assert(I_1)$, $assert(D_1)$, where $I_1 \oplus D_1$ is another EAP, we obtain the equivalent updated Evolp action program $(I \cup I_1) \oplus D \oplus D_1$ Formally:

Theorem 4. *Let* $I \oplus D_1 \oplus \ldots \oplus D_k$ *be any update EAP over* $(\mathcal{F}, \mathcal{A})$*. Let* $\bigoplus E_i^n$ *be a sequence of events such that:* $E_1 = K_1 \cup s$ *where* s *is any state of the world and* K *is any set of actions and the others* $E_i s$ *are: any set of actions* K_α*, or any set* ***initialize(***$\mathcal{F}_\beta$***)*** *where* $\mathcal{F}_\beta \subseteq \mathcal{F}$*, or any* D_i *with* $1 \leq i \leq k$*. Then* s' *is a resulting state from* s *given* $I \oplus D_1 \oplus \ldots \oplus D_k$ *and the sequence of sets of actions* $\bigoplus K_\alpha$ *iff there exists an evolving stable model* $M_1, \ldots, M_n$ *of* $I \oplus D$ *with event sequence* $\bigoplus E_i^n$ *such that* $M_n \equiv_{\mathcal{F}} s$

For instance, the updates to the original EAP of the example in this section are equivalent to the updated EAP is $I_{sum} \oplus D \oplus D_1 \oplus D_2$ such that $I_{sum} \equiv I \cup I_1 \cup I_2$ where I and D are as in example of section 3.1 and the Ii_s and Di_s are as in the description above.

Yet one more possibility opened by updated Evolp action programs is to cater for successive elaborations of a program. Consider an initial problem described by an EAP $I \oplus D$. If we want to describe an elaboration of the program, instead of *rewriting* $I \oplus D$ we can simply *update* it with new rules. This gives a new answer to the problem of elaboration tolerance [24] and also open the new possibility of *automatically update* action programs by other action programs.

The possibility to elaborate an action program is also discussed in [15] in the context of the $\mathcal{C}+$ language. The solution proposed is to consider $\mathcal{C}+$ programs whose rules have one extra fluent atom in their bodies that are assumed false by default. The elaboration of an action program P is the program $P \cup U$ where U is a new action program. The rules in U can defeat the rules in P by changing the truth value of the extra literals in their bodies. An advantage of EAP is that, in this framework, the possibility of updating rules is a built-in feature rather

then a programming technique involving manipulation of rules and introduction of new fluents. Moreover, in EAPs we can simply encode the new behaviours of the domain by new rules and then let these new rules update the previous ones.

6 Conclusions and future work

In this paper we have explored the possibility of using logic programs updates languages as action description languages. In particular we have focused our attention on the Evolp language. As a first point, we have defined a new action language paradigm, christened Evolp action programs, defined as a macro language over Evolp. We have provided an example of usage of such language. We have compared Evolp action programs with action languages $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ and provided simple translations into Evolp of these languages. Moreover we have shown, both by theoretical argumentation and practical examples, how some expressive capabilities of EAPs seem to be not replicable in these languages. Finally we have also shown and argued about the capability of $(\oplus P_i^m, \bigoplus E_i)$ to handle changes in the domain during the execution of actions.

Several important topics are not touched here, and will be subject of future work. An important field of research is how to deal, in the Evolp context, with the problem of planning [23]. Yet another topic involves the possibility of concurrent execution of actions. $EAPs$ allow the this possibility, nevertheless, we have not fully explored this topic, and confronted the results with extant works [6, 17]. Finally EAPs have to be implemented and tested in real and complex contexts.

References

1. J. J. Alferes, F. Banti, A. Brogi, and J. A. Leite. Semantics for dynamic logic programming: a principled based approach. In *Proceedings of the 7th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR-7)*, volume 1730 of *LNAI*, Berlin, 2004. Springer.
2. J. J. Alferes, A. Brogi, J. A. Leite, and L. M. Pereira. Evolving logic programs. In S. Flesca, S. Greco, N. Leone, and G. Ianni, editors, *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA'02)*, volume 2424 of *LNAI*, pages 50–61. Springer-Verlag, 2002.
3. J. J. Alferes, J. A. Leite, L. M. Pereira, H. Przymusinska, and T. C. Przymusinski. Dynamic updates of non-monotonic knowledge bases. *The Journal of Logic Programming*, 45(1–3):43–70, September/October 2000.
4. J. J. Alferes, L. M. Pereira, H. Przymusinska, and T. Przymusinski. LUPS: A language for updating logic programs. *Artificial Intelligence*, 132(1 & 2), 2002.
5. J. J. Alferes, L. M. Pereira, T. Przymusinski, H. Przymusinska, and P. Quaresma. Preliminary exploration on actions as updates. In M. C. Meo and M. V. Ferro, editors, *Proceedings of the 1999 Joint Conference on Declarative Programming (AGP-99)*, 1999.
6. C. Baral and M. Gelfond. Reasoning about effects of concurrent actions. *Journal of Logic Programming*, 31:85–118, 1997.

7. C. Baral, M. Gelfond, and Alessandro Provetti. Representing actions: Laws, observations and hypotheses. *Journal of Logic Programming*, 31, April–June 1997.
8. F. Buccafurri, W. Faber, and N. Leone. Disjunctive logic programs with inheritance. In D. De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming (ICLP-99)*, Cambridge, November 1999. MIT Press.
9. T. Eiter, M. Fink, G. Sabbatini, and H. Tompits. On properties of semantics based on causal rejection. *Theory and Practice of Logic Programming*, 2:711–767, November 2002.
10. T. Either, M. Fink, G. Sabbatini, and H. Tompits. A framework for declarative update specifications in logic programs. In Bernhard Nebel, editor, *Proceedings of the seventeenth International Conference on Artificial Intelligence (IJCAI-01)*, pages 649–654, San Francisco, CA, 2001. Morgan Kaufmann Publishers, Inc.
11. M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. A. Bowen, editors, *5th International Conference on Logic Programming*, pages 1070–1080. MIT Press, 1988.
12. M. Gelfond and V. Lifschitz. Representing actions and change by logic programs. *Journal of Logic Programming*, 17:301–322, 1993.
13. M. Gelfond and V. Lifschitz. Action languages. *Electronic Transactions on AI*, 16, 1998.
14. E. Giunchiglia, J. Lee, V. Lifschiz, N. Mc Cain, and H. Turner. Representing actions in logic programs and default theories: a situation calculus approach. *Journal of Logic Programming*, 31:245–298, 1997.
15. E. Giunchiglia, J. Lee, V. Lifschiz, N. McCain, and H. Turner. Nonmonotonic causal theories. *Artificial Intelligence*, 2003.
16. E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *AAAI'98*, pages 623–630, 1998.
17. J. Lee and V. Lifschitz. Describing additive fluents in action language C+. In William Nebel, Bernhard; Rich, Charles; Swartout, editor, *Proc. IJCAI-03*, pages 1079–1084, Cambridge, MA, 2003. To Appear.
18. J. A. Leite. *Evolving Knowledge Bases*, volume 81 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2003.
19. J. A. Leite and L. M. Pereira. Generalizing updates: from models to programs. In *LPKR'97: workshop on Logic Programming and Knowledge Representation*, 1997.
20. V. Lifschitz. *The Logic Programming Paradigm: a 25-Year Perspective*, chapter Action languages, answer sets and planning, pages 357–373. Springer Verlag, 1999.
21. V. Lifschitz and T. Woo. Answer sets in general non-monotonic reasoning (preliminary report). In B. Nebel, C. Rich, and W. Swartout, editors, *Proceedings of the 3th International Conference on Principles of Knowledge Representation and Reasoning (KR-92)*. Morgan-Kaufmann, 1992.
22. John Wylie Lloyd. *Foundations of Logic Programming*. Springer,, Berlin, Heidelberg, New York,, 1987.
23. J. McCarthy. Programs with commons sense. In *Proceedings of Teddington Conference on The Mechanization of Thought Process*, pages 75–91, 1959.
24. J. McCarthy. *Mathematical logic in artificial intelligence*, pages 297–311. Daedalus, 1988.
25. S. Russel and P. Norvig. *Artificial Intelligence A Modern Approach*, page 4. Artificial Intelligence. Prentice Hall, 1995.
26. D. McDermott S. Hanks. Nonmonotonic logic and temporal projection. *Artificial Intelligence*, 33:379–412, (1987).

Reasoning about logic-based agent interaction protocols

Matteo Baldoni, Cristina Baroglio,
Alberto Martelli, and Viviana Patti

Dipartimento di Informatica — Università degli Studi di Torino
C.so Svizzera, 185, I-10149 Torino (Italy)
E-mail: {baldoni,baroglio,mrt,patti}@di.unito.it

Abstract. The skill of reasoning about interaction protocols is very useful in many situations in the application framework of agent-oriented software engineering. In particular, we will tackle the cases of protocol selection, composition and implementation conformance w.r.t. an AUML sequence diagram. This work is based on DyLOG, an agent language based on modal logic that allows the inclusion, in the agent specification, of a set of interaction protocols.

1 Introduction

In Multi-Agent Systems (MASs) the communicative behavior of the agents plays a very important role, because it is the means by which agents cooperate for achieving a common goal or for competing for a resource. In order to rule communication, a set of shared protocols is commonly used. One of the most successful languages for designing them is AUML (Agent UML) [21]. This language is intuitive and easy to use for sketching the interactive behavior of a set of agents.

Our claim is that MAS engineering systems should encompass ways for obtaining declarative representations of protocols. The reason is that the use of declarative languages for protocol specification, although may be less intuitive, has the advantage of allowing the use of reasoning techniques in tasks like protocol validation or the verification of properties of the conversations within the system [13]. For instance, in [6] we proposed a logical framework, based on modal logic, that allows to include in an agent specification also a set of communication protocols. In this framework it is possible to reason about the effects of engaging specific conversations and to verify *properties* of the protocol: we can plan a conversation for achieving a particular goal, which respects the protocol, by checking if there is an execution trace of the protocol, after which the goal is satisfied. We can also verify if the composition of some protocols respects some desired constraint. Moreover, in [5, 7] we have shown how reasoning about conversation protocols can be used in an open application context where a personal assistant uses reasoning techniques for customizing the selection and the composition of web services.

The ability of reasoning about the properties of the interactions that occur among agents before they actually occur, may also by applied to support a MAS

developer during the design phase of the MAS. For instance, the developer could be supported in the selection of already developed protocols from a library and in the verification of compositional properties. Another crucial problem, typical of this application framework and that we think could be tackled by means of reasoning techniques, is checking the *conformance* of a logic agent or of a protocol implementation to the specification given in AUML during the system design phase. Broadly speaking an agent is conformant to a given protocol if its behavior is always legal w.r.t. the protocol; more precisely conformance verification is interpreted as the problem of checking that an agent never performs any dialogue move that is not foreseen by the AUML specification.

In particular, in this work we survey over different problems that could be tackled by means of reasoning techniques: we sketch how protocol conformance could be verified, how by applying reasoning techniques it is possible to select a protocol from a catalogue of available AUML diagrams, and also how we could deal with issues arising from the composition of various protocols in the MAS.

The work is organized as follows. First of all we will briefly describe how speech acts and protocols can be represented in the DyLOG agent programming language. In Sections 3 and 4 we will describe some major issues inherent agent-oriented software engineering, and we will show by means of examples how reasoning techniques can be adopted for solving such problems. Conclusions follow.

2 Specification of interaction protocols in DyLOG

Logic-based executable agent specification languages have been deeply investigated in the last years [20]. In this section we will briefly introduce DyLOG, a high-level logic programming language for modeling and programming rational agents, based on a modal theory of actions and mental attitudes where modalities are used for representing actions as well as beliefs for modeling the agent's mental state [10, 6]. It accounts both for atomic and complex actions, or procedures. Atomic actions are either world actions, affecting the world, or mental actions, i.e. sensing and communicative actions which only affect the agent beliefs. Complex actions are defined through (possibly recursive) definitions, given by means of Prolog-like clauses and by making use of action operators like sequence, test and non-deterministic choice. The action theory allows to cope with the problem of reasoning about complex actions with incomplete knowledge and in particular to address the temporal projection and planning problem.

Intuitively DyLOG allows the specification of a rational agent that can reason about its own behavior, can choose a course of actions conditioned by its mental state, and can use sensors and communication for obtaining fresh knowledge. The language also allows agents to reason about their communicative behavior by means of techniques for proving existential properties of the kind: given a protocol and a set of desiderata, is there a specific conversation, that respects the protocol, which also satisfies the desired conditions? In the following we will

describe how the communicative behavior of an agent can be represented in DyLOG and we will sketch the applicable reasoning techniques.

The DyLOG language supports communication both at the level of primitive speech acts and at the level of interaction protocols. Following the mentalistic approach, *speech acts* are considered as atomic actions, described in terms of preconditions and effects on the agent mental state, of form $\mathsf{speech_act}(ag_i, ag_j, l)$, where ag_i (sender) and ag_j (receiver) are agents and l (a fluent) is the object of the communication. Since speech acts can be seen as mental actions, affecting both the sender's and the receiver's mental state, we have modeled them by generalizing non-communicative action definitions, so to allow also the representation of the effects of an action executed by some other agent on the current agent mental state, described by a consistent set of *belief fluents*. Actually, in DyLOG each agent has a twofold, personal representation of the speech act: one is to be used when it is the sender, the other when it is the receiver. Such a representation provides the capability of *reasoning about* conversation effects from the subjective point of view of the agent holding the representation. In the speech act specification that holds when the agent is the sender, the preconditions contain some *sincerity condition* that must hold in its mental state. When it is the receiver, instead, the action is always executable. As an example, let us define the semantics of the *inform* speech act within the DyLOG framework:

a) $\Box(\mathcal{B}^{Self} l \wedge \mathcal{B}^{Self} \mathcal{U}^{Other} l \supset \langle \mathsf{inform}(Self, Other, l) \rangle \top)$
b) $\Box([\mathsf{inform}(Self, Other, l)] \mathcal{M}^{Self} \mathcal{B}^{Other} l)$
c) $\Box(\mathcal{B}^{Self} \mathcal{B}^{Other} authority(Self, l) \supset [\mathsf{inform}(Self, Other, l)] \mathcal{B}^{Self} \mathcal{B}^{Other} l)$
d) $\Box(\top \supset \langle \mathsf{inform}(Other, Self, l) \rangle \top)$
e) $\Box([\mathsf{inform}(Other, Self, l)] \mathcal{B}^{Self} \mathcal{B}^{Other} l)$
f) $\Box(\mathcal{B}^{Self} authority(Other, l) \supset [\mathsf{inform}(Other, Self, l)] \mathcal{B}^{Self} l)$

Clause (a) states that $Self$ will execute an inform act only if it believes l (we use the modal operator $\mathcal{B}^{ag_i}$ to model the beliefs of agent ag_i) and it believes that the receiver ($Other$) does not know l. It also considers possible that the receiver will adopt its belief (the modal operator $\mathcal{M}^{ag_i}$ is defined as the dual of $\mathcal{B}^{ag_i}$, intuitively $\mathcal{M}^{ag_i}\varphi$ means the ag_i considers φ possible), clause (b), although it cannot be certain about it -autonomy assumption-. If agent $Self$ believes to be considered a trusted *authority* about l by the receiver, it is also confident that $Other$ will adopt its belief, clause (c). Instead, when $Self$ is the receiver, the effect of an inform act is that $Self$ will believe that l is believed by the sender ($Other$), clause (e), but $Self$ will adopt l as an own belief only if it thinks that $Other$ is a trusted authority, clauses (f).

DyLOG supports also the development of *conversation protocols*, that build on individual speech acts and specify communication patterns guiding the agent communicative behavior during a protocol-oriented dialogue. Reception of messages is modeled as a special kind of sensing action, what we call *get message actions*. Indeed receiving a message is interpreted as a query for an external input, whose outcome is unpredictable. The main difference w.r.t. normal sensing actions is that *get message actions* are defined by means of speech acts performed

by the interlocutor. Protocols are thus expressed by means of a collection of procedure axioms of the action logic, having form $\langle p_0 \rangle \varphi \subset \langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_n \rangle \varphi$, where p_0 is the procedure name the p_i's can be i's communicative acts or special sensing actions for the reception of message. Each agent has a subjective perception of the communication with other agents, for this reason each protocol has as many procedural representations as the possible roles in the conversation. The importance of roles has been underlined also recently in works, such as [17].

Given a set $\Pi_{\mathcal{C}}$ of simple action laws defining an agent ag_i's primitive speech acts, a set $\Pi_{\mathcal{S}get}$ of axioms for the reception of messages, and a set $\Pi_{\mathcal{CP}}$, of procedure axioms specifying a collection of conversation protocols, we denote by CKit^{ag_i} (the *communication kit* of a DyLOG agent ag_i), the triple $(\Pi_{\mathcal{C}}, \Pi_{\mathcal{CP}}, \Pi_{\mathcal{S}get})$. CKit^{ag_i} is a part of Π_{ag_i}, i.e. the domain description of the agent ag_i, including also S_0, i.e the initial set of ag_i's belief fluents, and eventually laws and axioms for specifying the agent non communicative behaviors.

2.1 Reasoning about interaction protocols in DyLOG

Given a DyLOG domain description Π_{ag_i} containing a CKit^{ag_i} with the specifications of the interaction protocols and of the relevant speech acts, a *planning* activity can be triggered by *existential queries* of form $\langle p_1 \rangle \langle p_2 \rangle \ldots \langle p_m \rangle Fs$, where each p_k $(k = 1, \ldots, m)$ may be an atomic or complex action (a primitive speech act or an interaction protocol), executed by our agent, or an external[1] speech act, that belongs to CKit^{ag_i}. Checking if the query succeeds corresponds to answering to the question "is there an execution of p_1, ..., p_m leading to a state where the conjunction of belief fluents Fs holds for agent ag_i?". Such an execution is a plan to bring about Fs. The procedure definition constrains the search space. During the planning process get_message actions are treated as sensing actions, whose outcome cannot be predicted before the actual execution: since agents cannot read each other's mind, they cannot know in advance the answers that they will receive.

Depending on the task that one has to execute, it may alternatively be necessary to take into account all of the possible alternatives (which, we can foresee them because of the existence of the protocol) or just to find one of them. In the former case, the extracted plan will be *conditional*, in the sense that for each get_message and for each sensing action it will contain as many branches as possible action outcomes. Each path in the resulting tree is a linear plan that brings about Fs. In the latter case, instead, the plan is linear.

3 Reasoning about protocols in MAS design

Generally speaking MASs are made of heterogeneous agents, which have different ways of representing their knowledge and adopt different mechanisms for

[1] By the word *external* we denote a speech act in which our agent plays the role of the receiver.

reasoning about it. Despite heterogeneity, agents need to interact and exchange information in order to cooperate or compete for the control of shared resources. This is obtained by means of interaction protocols, which result in being the *connective tissue* of the system. MAS engineering systems support the design of interaction protocols by means of graphical editors for the AUML language.

The AUML specification of a protocol is obtained by means of sequence diagrams [21], in which the interactions among the participants are modeled as message exchange and are arranged in time sequences. Sequence diagrams have two dimensions: the vertical (time) dimension specifying when a message is sent or expected, and the horizontal dimension that expresses the participants and their different roles. This kind of representation is very high-level and usually needs to be further specified in order to arrive to real implementations. In fact, the semantics of the atomic speech acts is not given by AUML; at implementation time, depending on the chosen ontology of speech acts, it may be necessary to express constraints or preconditions that depend on the agent mental state, that are not reported in the sequence diagrams. Since AUML sequence diagrams do not represent complete programs, it is not possible to automatically translate them in a way that fully expresses the communicative behavior of one or more agents in the application scenario. The protocol is to be *implemented*. On the other hand, given a protocol implementation it would be nice to have the possibility of automatically verifying its conformance to the desired protocol.

As mentioned in the introduction, a program is conformant to a protocol if all the message exchanges that it produces are foreseen by the protocol. The adoption of a logic formalism for implementing the protocols greatly simplifies this kind of verification, as we will see in the case of DyLOG in the next section. Differently of [11], the conformance property will be expressed from a language-theoretic point of view instead of from a logic point of view, by interpreting the problem of conformance verification as a problem of inclusion of a context-free language (CFL) into a regular language. In this process, the particular form of axiom, namely *inclusion axiom*, used to define protocol clauses in a DyLOG implementation, comes to help us. These axioms have interesting computational properties because they can be considered as *rewriting rules* [9, 3]. In [12] this kind of axioms is used for defining *grammar logics* and some relations between formal languages and such logics are analyzed.

On the side of protocol implementation, logic languages for reasoning about action and change, like DyLOG, seem particularly suitable. The reason is that although AUML accounts for a fast and intuitive prototyping in a graphical environment, it does not straightforwardly allow the *proof of properties* of the resulting system. Nevertheless, given the crucial role that protocols play, the ability of *reasoning about properties* of the interactions, occurring among the agents, is a key stone of the design and engineering of agent systems. By using DyLOG (or a similar language) it is possible to use the reasoning techniques embedded in the language for executing various tasks. Thus, besides the verification of the conformance of an implementation to a protocol, other possible applications of reasoning techniques include the intelligent use of libraries. In fact, it is not likely

that the designer of a MAS redesigns every time new protocols, instead, he/she will more likely use *catalogues* of available protocols, in which searching for one of interest (notice that search could be based on the concept of conformance). In this case, the designer will need to know *before* the actual implementation of the agent, if the selected protocol fits some specific requisites of the system that is being developed. This problem can be interpreted as the problem of verifying if it is possible that a property of interest holds after the execution of the protocol in a given initial state. In this line, the designer may be interested in verifying if the *composition* of a set of protocols will be useful in the current application.

These observations are particularly relevant in the development of MAS prototyping systems (e.g. DCaseLP [19, 1]): in fact, a system which supports the design of the (communicative) interaction between the agents in a MAS only by means of AUML sequence diagrams, can carry on verifications only by animating the system and checking what happens in specific cases. A solution that has clear limitations since we get information only about the cases that have been tried. The worst limitation is that it is not possible to return to the system engineer the assumptions under which a goal is achieved or a property holds after the interaction.

4 Reasoning about protocols: some examples

Let us now illustrate the usefulness of reasoning techniques in the application domain of MAS design by means of examples. To this aim, consider this scenario: a MAS designer must develop a set of interaction protocols for a restaurant and a cinema that, for promotional reasons, will cooperate as described hereafter. A customer that makes a reservation at the restaurant will get a free ticket for a movie shown by the cinema. By restaurant and cinema we do not mean a specific restaurant or cinema but a generic service provider that will interact with its customers according to the protocol. Figure 1 reports an example of protocols, designed in AUML, for the two families of providers; the protocol described by (iii) is followed by the restaurant while the protocol described by (i) and (ii) is followed by the cinema. Let us describe the translation of the two protocols into a DyLOG representation. Observe that each of them has two complementary views: the view of the customer and the view of the provider. In this example we report only the view of the customer, which is what we need for the reasoning process. In the following, the subscripts next to the protocol names are a writing convention for representing the role that the agent plays; so, for instance, Q stands for *querier*, C for *customer*, and so on. The restaurant protocol is the following:

(a) $\langle \mathsf{reserv_rest_1}_C(Self, Service, Time)\rangle\varphi \subset$
$\quad\langle \mathsf{yes_no_query}_Q(Self, Service, available(Time))$;
$\quad\quad\mathcal{B}^{Self} available(Time)?$;
$\quad\quad\mathsf{get_info}(Self, Service, reservation(Time))$;
$\quad\quad\mathsf{get_info}(Self, Service, cinema_promo)$;
$\quad\quad\mathsf{get_info}(Self, Service, ft_number)\rangle\varphi$

(b) $[\mathsf{get_info}(Self, Service, Fluent)]\varphi \subset [\mathsf{inform}(Service, Self, Fluent)]\varphi$

Procedure (a) describes the customer-view of the restaurant protocol. The customer asks if a table is available at a certain time, if so, the restaurant informs the customer that a reservation has been taken and, also, it informs the customer that it gained a promotional free ticket for a cinema (*cinema_promo*) and it returns a code number (*ft_number*). Clause (b) shows how get_info can be implemented as an inform act executed by the service and having as recipient the customer. In the DyLOG syntax the question mark corresponds to checking the value of a fluent in the current state while the semicolon is the sequencing operator of two actions. The cinema protocol, instead is:

(c) $\langle \mathsf{reserv_cinema_1}_C(Self, Service, Movie)\rangle\varphi \subset$
$\langle \mathsf{yes_no_query}_Q(Self, Service, available(Movie))$;
$\mathcal{B}^{Self} available(Movie)?$;
$\mathsf{yes_no_query}_I(Self, Service, cinema_promo)$;
$\neg\mathcal{B}^{Self} cinema_promo?$;
$\mathsf{yes_no_query}_I(Self, Service, pay_by(c_card))$;
$\mathcal{B}^{Self} pay_by(c_card)?$;
$\mathsf{inform}(Self, Service, cc_number)$;
$\mathsf{get_info}(Self, Service, reservation(Movie))\rangle\varphi$

(d) $\langle \mathsf{reserv_cinema_1}_C(Self, Service, Movie)\rangle\varphi \subset$
$\langle \mathsf{yes_no_query}_Q(Self, Service, available(Movie))$;
$\mathcal{B}^{Self} available(Movie)?$;
$\mathsf{yes_no_query}_I(Self, Service, cinema_promo)$;
$\mathcal{B}^{Self} cinema_promo?$;
$\mathsf{inform}(Self, Service, ft_number)$;
$\mathsf{get_info}(Self, Service, reservation(Movie))\rangle\varphi$

Supposing that the desired movie is available, the cinema alternatively accepts credit card payments (c) or promotional tickets (d).

4.1 Conformance

Supposing that the designer produced the AUML sequence diagrams reported in Figure 1, and, then, implemented them in DyLOG, the first problem to solve is to check the conformance of the implementation w.r.t. the diagrams. For the sake of simplicity, we will sketch the method that we mean to follow focusing on one of the drawn protocols: the reserv_cinema_1 protocol.

In order to verify the conformance of a DyLOG interaction protocol to an AUML interaction protocol, that the DyLOG program is supposed to implement, we represent the AUML sequence diagram as a *formal language* by means of a *grammar*. More precisely, it is quite easy to see that, given a sequence diagram, it is possible to represent the set of the possible dialogues by means of a *regular*

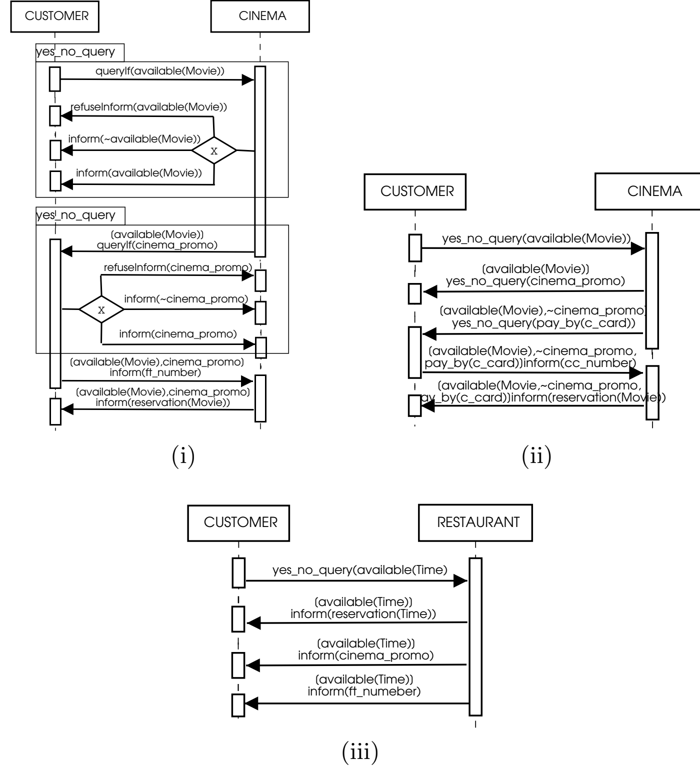


Fig. 1. AUML interaction protocols representing the interactions between the customer and the provider: (i) and (ii) are followed by the cinema service, (iii) is followed by the restaurant. Formulas among square brackets represent conditions on the execution of the speech act.

grammar, whose set of atoms corresponds to the set of atomic speech acts. For example, let us consider the reserv_cinema_1 protocol (see Figure 1 (i)), the following grammar $G_{\mathsf{reserv_cinema_1}}$ represents it[2]:

$$Q_0 \longrightarrow \mathsf{queryIf}(customer, cinema, available(Movie))\ Q_1$$
$$Q_1 \longrightarrow \mathsf{refuseInform}(cinema, customer, available(Movie))$$
$$Q_1 \longrightarrow \mathsf{inform}(cinema, customer, \neg available(Movie))$$
$$Q_1 \longrightarrow \mathsf{inform}(cinema, customer, available(Movie))\ Q_2$$
$$Q_2 \longrightarrow \mathsf{queryIf}(cinema, customer, cinema_promo)\ Q_3$$
$$Q_3 \longrightarrow \mathsf{refuseInform}(customer, cinema, cinema_promo)$$
$$Q_3 \longrightarrow \mathsf{inform}(customer, cinema, \neg cinema_promo)$$
$$Q_3 \longrightarrow \mathsf{inform}(customer, cinema, cinema_promo)\ Q_4$$
$$Q_4 \longrightarrow \mathsf{inform}(customer, cinema, ft_number)\ Q_5$$

[2] Details about the translation from AUML to the grammar can be found in [8].

$Q_5 \longrightarrow \mathsf{inform}(cinema, customer, reservation(Movie))$

By means of $L(G_{\mathsf{reserv_cinema_1}})$ we denote the language generated by it. Intuitively, it represents all the *legal* conversations.

Now, we are in the position to give our first definition of conformance: the *agent conformance*.

Definition 1 (Agent conformance). *Let $D = (\Pi, \mathsf{CKit}^{ag_i}, S_0)$ be a domain description, $\mathsf{p}_{dylog} \in \mathsf{CKit}^{ag_i}$ be an implementation of the interaction protocol p_{AUML} defined by means of an AUML sequence diagram. Moreover, let us define the set $\Sigma(S_0)$ as $\{\sigma | (\Pi, \mathsf{CKit}^{ag_i}, S_0) \vdash_{ps} \langle \mathsf{p}_{dylog} \rangle \top \text{ w. a. } \sigma\}$. We say that the agent described by means of D is* conformant w.r.t. the sequence diagram p_{AUML} *if and only if:*

$$\Sigma(S_0) \subseteq L(G_{\mathsf{p}_{AUML}}) \quad (1)$$

In other words, the agent conformance property holds if we can prove that every conversation, that is an instance of the protocol implemented in our language (an execution trace of p_{dylog}), is a legal conversation according to the grammar that represents the AUML sequence diagram p_{AUML}; that is to say, that conversation is also generated by the grammar $G_{\mathsf{p}_{AUML}}$.

The agent conformance depends on the initial state S_0. Different initial states can determine different possible conversations (execution traces). A notion of agent conformance, that is independent from the initial state, can also be defined:

Definition 2 (Agent strong conformance). *Let $D = (\Pi, \mathsf{CKit}^{ag_i}, S_0)$ be a domain description, let $\mathsf{p}_{dylog} \in \mathsf{CKit}^{ag_i}$ be an implementation of the interaction protocol p_{AUML} defined by means of an AUML sequence diagram. Moreover, let us define the set $\Sigma = \bigcup_S \Sigma(S)$, where S ranges over all possible initial states. We say that the agent described by means of D is* strongly conformant w.r.t. the sequence diagram p_{AUML} *if and only if:*

$$\Sigma \subseteq L(G_{\mathsf{p}_{AUML}}) \quad (2)$$

In other words, the agent strong conformance property holds if we can prove that every conversation for every possible initial state is a legal conversation, i.e. it is also generated by the grammar that represents the AUML sequence diagram. It is easy to see that agent strong conformance (2) implies agent conformance (1).

Agent strong conformance, differently than agent conformance, does not depend on the initial state but it still depends on the set of speech acts defined in CKit^{ag_i}. A stronger notion of conformance should require that a DyLOG implementation is conformant w.r.t. an AUML sequence diagram independently of the semantics of the speech acts. In other world, we would like to prove a sort of "structural" conformance of the implemented protocol w.r.t. the corresponding AUML sequence diagram. In order to do this, we define a formal grammar from the DyLOG implementation of a conversation protocol. In this process, the particular form of axiom, namely *inclusion axiom*, used to define protocol clauses in a DyLOG implementation, comes to help us. Actually, such axioms have a natural interpretation as rewriting rules [2, 12].

Given a domain description $(\Pi, \mathsf{CKit}^{ag_i}, S_0)$ and an conversation protocol $\mathsf{p}_{dylog} \in \mathsf{CKit}^{ag_i} = (\Pi_{\mathcal{C}}, \Pi_{\mathcal{CP}}, \Pi_{\mathcal{S}get})$, we define the grammar $G_{\mathsf{p}_{dylog}} = (T, V, P, S)$, where T is the set of all terms that define the set of speech acts in $\Pi_{\mathcal{C}}$, V is the set of all the terms that define a conversation protocol or a get message action in $\Pi_{\mathcal{CP}}$ or $\Pi_{\mathcal{S}get}$. P is the set of production rules of the form $p_0 \longrightarrow p_1 p_2 \dots p_n$ where $\langle p_0 \rangle \varphi \supset \langle p_1 \rangle \langle p_2 \rangle \dots \langle p_n \rangle \varphi$ is an axiom that defines either a conversation protocol (that belongs to $\Pi_{\mathcal{CP}}$) or a get message action (that belongs to $\Pi_{\mathcal{S}get}$). Note that test actions $\langle Fs? \rangle$ are not reported in the production rules. Finally, the start symbol S is the symbol p_{dylog}. Let us define $L(G_{\mathsf{p}_{dylog}})$ as the language generated by means of the grammar $G_{\mathsf{p}_{dylog}}$. It is easy to see that $L(G_{\mathsf{p}_{dylog}})$ is a context-free language since $G_{\mathsf{p}_{dylog}}$ is a context-free grammar. Intuitively, the language $L(G_{\mathsf{p}_{dylog}})$ represents all the possible sequences of speech acts (conversations) allowed by the DyLOG protocol p_{dylog} independently of the evolution of the mental state of the agent.

Definition 3 (Protocol conformance). *Let $D = (\Pi, \mathsf{CKit}^{ag_i}, S_0)$ be a domain description, let $\mathsf{p}_{dylog} \in \mathsf{CKit}^{ag_i}$ be an implementation of the interaction protocol p_{AUML} defined by means of an AUML sequence diagram. We say that p_{dylog} is* conformant to the sequence diagram p_{AUML} *if and only if:*

$$L(G_{\mathsf{p}_{dylog}}) \subseteq L(G_{\mathsf{p}_{AUML}}) \tag{3}$$

It is possible to prove that *protocol conformance* (3) implies *agent strong conformance* (2).

In this case, it is straightforward to prove that for the customer view $\mathsf{reserv_cinema_1}_C$ the protocol conformance holds w.r.t. protocol reserv_cinema_1. It is worth noting the following property of the protocol conformance.

Proposition 1. *Protocol conformance is decidable.*

Proof. Equation (3) is equivalent to $L(G_{\mathsf{p}_{dylog}}) \cap \overline{L(G_{\mathsf{p}_{AUML}})} = \emptyset$. Now, $L(G_{\mathsf{p}_{dylog}})$ is a context-free language while $L(G_{\mathsf{p}_{AUML}})$ is a regular language. Since the complement of a regular language is still regular, $\overline{L(G_{\mathsf{p}_{AUML}})}$ is a regular language. The intersection of a context-free language and a regular language is a context-free language. For context-free languages, the emptiness is decidable [15].

Proposition 1 tells us that an algorithm for verifying protocol conformance exists. However, we also have a straightforward methodology for implementing protocols in a way that conformance w.r.t. the AUML specification is respected. In fact, we can build our implementation starting from the grammar $G_{\mathsf{p}_{AUML}}$, and applying the inverse of the process that we described for passing from a DyLOG implementation to the grammar $G_{\mathsf{p}_{dylog}}$. In this way we obtain a skeleton of a DyLOG implementation of p_{AUML} that is to be completed by adding the desired ontology for the speech acts and customized with tests. Such an implementation trivially satisfies protocol conformance and, then, all the other degrees of conformance defined above.

4.2 Composition

One example in which it is useful to reason about protocol composition, is the situation in which the same customer is supposed to interact with the restaurant and the cinema providers, one after the other. In fact, the developer must be sure that the customer, by interacting with the composition (by sequentialization) of the two protocols, will obtain what desired. In particular, suppose that the developer wants to verify if the protocols that he/she defined allow the following interaction: it is possible to make a reservation at the restaurant and, then, at the cinema, taking advantage of the promotion. Let us consider the query:

$$\langle \mathsf{reserv_rest_1}_C(customer, restaurant, dinner)\ ;\ \mathsf{reserv_cinema_1}_C(customer, cinema, movie)\rangle (\mathcal{B}^{customer} cinema_promo \wedge \mathcal{B}^{customer} reservation(dinner) \wedge \mathcal{B}^{customer} reservation(movie) \wedge \mathcal{B}^{customer}\mathcal{B}^{C} ft_number)$$

that amounts to determine if it is possible to compose the interaction so to reserve a table for dinner ($\mathcal{B}^{customer} reservation(dinner)$) and to book a ticket for the movie *movie* ($\mathcal{B}^{customer} reservation(movie)$), exploiting a promotion ($\mathcal{B}^{customer} cinema_promo$). The obtained free ticket is to be spent ($\mathcal{B}^{customer}$ $\mathcal{B}^{cinema}$ ft_number), i.e. *customer* believes that after the conversation the chosen cinema will know the number of the ticket given by the selected restaurant.

In the case in which the customer has neither a reservation for dinner nor a reservation for the cinema or a free ticket, the query succeeds, returning the following linear plan:

$\mathsf{queryIf}(customer, restaurant, available(dinner))$;
$\boxed{\mathsf{inform}(restaurant, customer, available(dinner))\ ;}$
$\mathsf{inform}(restaurant, customer, reservation(dinner))$;
$\mathsf{inform}(restaurant, customer, cinema_promo)$;
$\mathsf{inform}(restaurant, customer, ft_number)$;
$\mathsf{queryIf}(customer, cinema, available(movie))$;
$\boxed{\mathsf{inform}(cinema, customer, available(movie))\ ;}$
$\mathsf{queryIf}(cinema, customer, cinema_promo)$;
$\mathsf{inform}(customer, cinema, cinema_promo)$;
$\mathsf{inform}(customer, cinema, ft_number)$;
$\mathsf{inform}(cinema, customer, reservation(movie))$

This means that there is first a conversation between *customer* and *restaurant* and, then, a conversation between *customer* and *cinema*, that are instances of the respective conversation protocols, after which the desired condition holds.

Notice that the linear plan will actually lead to the desired goal given that some assumptions about the provider's answers hold. In the above plan, assumptions have been outlined with a box. For instance, that a seat at the cinema is free. The difference with the other inform acts in the plan (from a provider to the customer) is that while for those the protocol does not offer any alternative, the

outlined ones correspond just to one of the possible answers foreseen by the protocol. In the example they are answers foreseen by a yes_no_query protocol (see Figure 1 (i) and [6]). The actual answer can be known only at execution time, however, thanks to the existence of the protocol, it is possible to understand the conditions that lead to success.

4.3 Selection

Another situation in which the developer may need support is to search into a library of available policies for an interaction protocol that describes a service of interest and that is suitable to the application that he/she is designing. For instance, the developer must design a protocol for a restaurant that allows to make a reservation not necessarily using a credit card. In this case the developer will first search the library of available protocols looking for those that satisfy this request. This search cannot be accomplished only based on descriptive keywords but requires a form of reasoning on the way in which the interaction is carried on. Suppose that *search_service* is a procedure that allows one to search into a library for a protocol in a given category; then, the query would look like:

$$\langle search_service(restaurant, Protocol)\ ;\ Protocol(customer, service, time)\rangle$$
$$(\mathcal{B}^{customer}\neg\mathcal{B}^{service} cc_number \wedge \mathcal{B}^{customer} reservation(time))$$

which means: look for a protocol that has one possible execution, after which the service provider does not know the customer's credit card number, and a reservation has been taken.

5 Conclusions and related work

In this paper we have proposed a logic-based approach to agent interaction protocol specification and we have shown the advantages of using reasoning techniques based on such a logical formalization, especially in the context of agent-oriented software engineering (AOSE). We used as agent language DyLOG, a high-level logic programming language for modeling and programming rational agents, based on a modal theory of actions and mental attitudes, that allows to include in the agent specification a set of interaction protocols. The DyLOG language allows reasoning about the effects of engaging specific conversations. By doing so, an agent can plan a conversation for achieving a particular goal, that respects the protocol, while the system designer can exploit the same reasoning techniques to select a protocol from a library or to verify if the composition of a set of given protocols respects some desired constraint.

Moreover, we have shown that our logical representation of protocols allows us to deal with the matter of checking the agent conformance w.r.t. a protocol represented as a AUML interaction diagram. This is a crucial problem to face in an AOSE perspective and it can be considered as a part of the process of engineering interaction protocols sketched in [16]. Indeed, supposing to have both an

AUML sequence diagram, which formally specifies a protocol, and an implementation of the same protocol in DyLOG, a key problem to solve is checking the conformance of the implementation to the diagram. In fact, when protocols are implemented in DyLOG they become part of the agent communication policies, which are usually determined by the agent mental state. What we check is if this strategy is conformant to the AUML protocol, at least in the sense that the agent never performs any illegal dialogue move. In other words we prove a sort of "structural" conformance of the implemented protocol w.r.t the specification.

The problem of checking the agent conformance to a protocol in a logical framework has been faced also in [11]. In a broad sense our notion of conformance bears along some similarities with the notion of agent weak conformance, introduced in this work. In [11] agent communication strategies and protocol specification are both represented by means of sets of *if-then rules* in a logic-based language, which relies on abductive logic programming. A notion of weak conformance is introduced, which allows to check if the possible moves that an agent can make, according to a given communication strategy, are legal w.r.t. the protocol specification. The conformance test is done by abstracting from (i.e. disregarding) any condition related to the agent private knowledge, which is not considered as relevant in order to decide weak conformance, although it could, actually, prevent an agent from performing a particular move.

Our framework allows us to give a finer notion of conformance, for which we can distinguish different degrees of abstraction with respect to the agent private mental state. It allows us to define in an elegant manner which parts of a protocol implementation must fit the protocol specification and to describe in a modular way how the protocol implementation can be enriched with respect to the protocol specification, without compromising the conformance. Such an enrichment is important when using logic agents, that support sophisticated forms of reasoning. Indeed, the ability of reasoning about the properties of the interactions that occur among agents before they actually occur, may by applied to support a MAS developer during the design phase. We have recently begun to study [4] the methodological and physical integration of DyLOG into the DCaseLP [1, 18] MAS prototyping environment following the what already done for integrating tuProlog [14]. The aim of this work is to enrich the DCaseLP framework with the ability of reasoning about AUML interaction protocols thanks to a translation from AUML to DyLOG. In this context it will be extremely useful to have formal methods for proving conformance properties.

Acknowledgement

This research is partially supported by MIUR Cofin 2003 "Logic-based development and verification of multi-agent systems" national project.

References

1. E. Astesiano, M. Martelli, V. Mascardi, and G. Reggio. From Requirement Specification to Prototype Execution: a Combination of a Multiview Use-Case Driven

Method and Agent-Oriented Techniques. In J. Debenham and K. Zhang, editors, *Proceedings of the 15th International Conference on Software Engineering and Knowledge Engineering (SEKE'03)*, pages 578–585. The Knowledge System Institute, 2003.

2. M. Baldoni. *Normal Multimodal Logics: Automatic Deduction and Logic Programming Extension*. PhD thesis, Dipartimento di Informatica, Università degli Studi di Torino, Italy, 1998. Available at `http://www.di.unito.it/~baldoni/`.
3. M. Baldoni. Normal Multimodal Logics with Interaction Axioms. In D. Basin, M. D'Agostino, D. M. Gabbay, S. Matthews, and L. Viganò, editors, *Labelled Deduction*, volume 17 of *Applied Logic Series*, pages 33–53. Applied Logic Series, Kluwer Academic Publisher, 2000.
4. M. Baldoni, C. Baroglio, I. Gungui, A. Martelli, M. Martelli, V. Mascardi, V. Patti, and C. Schifanella. Reasoning about agents' interaction protocols inside dcaselp. In J. Leite, A. Omicini, P. Torroni, and P. Yolum, editors, *Proc. of the International Workshop on Declarative Language and Technologies, DALT'04*, New York, July 2004. To appear.
5. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about Conversation Protocols in a Logic-based Agent Language. In A. Cappelli and F. Turini, editors, *AI*IA 2003: Advances in Artificial Intelligence, 8th Congress of the Italian Association for Artificial Intelligence*, volume 2829 of *LNAI*, pages 300–311. Springer, September 2003.
6. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about self and others: communicating agents in a modal action logic. In C. Blundo and C. Laneve, editors, *Theoretical Computer Science, 8th Italian Conference, ICTCS'2003*, volume 2841 of *LNCS*, pages 228–241, Bertinoro, Italy, October 2003. Springer.
7. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about interaction protocols for web service composition. In M. Bravetti and G. Zavattaro, editors, *Proc. of 1st Int. Workshop on Web Services and Formal Methods, WS-FM 2004*, Electronic Notes in Theoretical Computer Science. Elsevier Science Direct, 2004.
8. M. Baldoni, C. Baroglio, A. Martelli, V. Patti, and C. Schifanella. Verifying protocol conformance for logic-based communicating agents. Submitted, 2004.
9. M. Baldoni, L. Giordano, and A. Martelli. A Tableau Calculus for Multimodal Logics and Some (Un)Decidability Results. In H. de Swart, editor, *Proc. of TABLEAUX'98*, volume 1397 of *LNAI*, pages 44–59. Springer-Verlag, 1998.
10. M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Programming Rational Agents in a Modal Action Logic. *Annals of Mathematics and Artificial Intelligence, Special issue on Logic-Based Agent Implementation*, 41(2-4), 2004. To appear.
11. Ulrich Endriss, Nicolas Maudet, Fariba Sadri, and Francesca Toni. Logic-based agent communication protocols. In F. Dignum, editor, *Advances in agent communication languages*, volume 2922 of *Lecture Notes in Artificial Intelligence (LNAI)*. Springer-Verlag, 2004. To appear.
12. L. Fariñas del Cerro and M. Penttonen. Grammar Logics. *Logique et Analyse*, 121-122:123–134, 1988.
13. F. Guerin and J. Pitt. Verification and Compliance Testing. In H.P. Huget, editor, *Communication in Multiagent Systems*, volume 2650 of *LNAI*, pages 98–112. Springer, 2003.
14. I. Gungui and V. Mascardi. Integrating tuProlog into DCaseLP to engineer heterogeneous agent systems. Parma, Italy, June 2004. In this volume.
15. J. E. Hopcroft and J. D. Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley Publishing Company, 1979.

16. M. P. Huget and J.L. Koning. Interaction Protocol Engineering. In H.P. Huget, editor, *Communication in Multiagent Systems*, volume 2650 of *LNAI*, pages 179–193. Springer, 2003.
17. S. Breuckner J. Odell, H. Van Dyke Parunak and M. Fleischer. Temporal aspects of dynamic role assignment. In J. Odell P. Giorgini, J. Müller, editor, *Agent-Oriented Software Engineering (AOSE) IV*, volume (forthcoming) of *LNCS*, Berlin, 2004. Springer.
18. M. Martelli and V. Mascardi. From UML diagrams to Jess rules: Integrating OO and rule-based languages to specify, implement and execute agents. In F. Buccafurri, editor, *Proceedings of the 8th APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP'03)*, pages 275–286, 2003.
19. M. Martelli, V. Mascardi, and F. Zini. Caselp: a prototyping environment for heterogeneous multi-agent systems. Available at http://www.disi.unige.it/person/MascardiV/Download/aamas-journal-MMZ04.ps.gz.
20. V. Mascardi, M. Martelli, and L. Sterling. Logic-Based Specification Languages for Intelligent Software Agents. *Theory and Practice of Logic Programming Journal*, 2004. to appear.
21. J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In *Proceedings of the Agent-Oriented Information System Workshop at the 17th National Conference on Artificial Intelligence*. 2000.

Contracts in Multiagent Systems: the Legal Institution Perspective

Guido Boella[1] and Leendert van der Torre[2]

[1]Dipartimento di Informatica - Università di Torino - Italy, guido@di.unito.it
[2]CWI - Amsterdam - The Netherlands, torre@cwi.nl

Abstract. In this paper we address the problem of how the autonomy of agents in an organization can be enhanced by means of contracts. Contracts are modelled as legal institutions: systems of legal rules which allow to change the regulative and constitutive rules of an organization. The methodology we use is to attribute to organizations mental attitudes, beliefs, desires and goals, and to take into account their behavior by using recursive modelling.

1 Introduction

One of the main challenges in multi-agent societies is the coordination of the autonomous agents. Coordination can be achieved finding a trade off between a bottom-up view and a top-down view of the problem. In the former, the MAS is an aggregation of autonomous agents interacting with each other, where their emergent behavior is not necessarily the desired one. In the latter, the system's objectives are achieved without requiring specific agent's internal design, by means of organizational design together with roles and norms as incentives for cooperation.

As Dignum *et al.* [12] note, however, the interaction structure of the organization should be not be completely fixed in advance. The autonomy of the agents should be preserved even if within limits. For this reason, some approaches like [10, 12, 19] introduce the possibility for agents to stipulate contracts. A contract can be defined as a statement of intent that regulates behavior among organizations and individuals. Contracts have been proposed as means to make explicit the way agents can change the interaction with and within the society: they create obligations, permissions and new possibilities of interactions. From a contractual perspective, organizations can be seen as the possible sets of agreements for satisfying the diverse interests of self interested individuals [10].

Social order, thus, emerges from the negotiation of contracts about the rights and duties of participants, rather than being given in advance. But the organization itself specifies the possible contracts and enforces the obligations created by them as they were issued by the organization itself. As Ruiter [20] shows, however, from the legal point of view, legal effects of actions of the members of a legal system (as an organization is) are a difficult problem. Contracts do not concern only the regulative aspects of a legislation (i.e., the rules of behavior specified in obligations), or the constitutive part of it (i.e., the rules introducing institutional facts such bidding in an auction). Rather, contracts are *legal institutions*: "systems of [regulative and constitutive] rules that provide

frameworks for social action within larger rule-governed settings" [20]; in our case the larger setting is represented by organizations.

This systemic view of legal institutions emerged only recently in legal studies [20], since legal positivism [15] mainly focused on the regulative aspects of law and its justification. For this reason is necessary to address the problem of contracts being aware of the peculiarities of legal institutions.

The research question of this paper is: how can be legal institutions, like contracts, be formalized? and, as subquestions, how can agents modify the behavior of the organization via contracts? Which games can agents play when they are allowed to make contracts?

As methodology we use the agent metaphor: we attribute to organizations mental attitudes, beliefs, desires and goals, and we take into account their behavior by using recursive modelling [13]. We apply to organizations the methodology we adopted for social entities like groups, virtual communities [4] and normative multiagent systems [3, 7, 6].

In the next section we discuss constitutive rules and how legal institutions are created. In Section 3 we discuss the conceptual model, with the definition of obligations and contracts. In Section 4 we present the games which can be played with contracts, together with a detailed example. Related work and summary close the paper.

2 Legal institutions

Normative multiagent systems, like organizations, are "sets of agents [...] whose interactions can be regarded as norm-governed; the norms prescribe how the agents ideally should and should not behave. [...] Importantly, the norms allow for the possibility that actual behavior may at times deviate from the ideal, i.e., that violations of obligations, or of agents' rights, may occur" (Jones and Carmo [16]).

In [3] we formalize the relation between multiagent systems and normative systems by attributing mental states to agents as well as to normative systems, as proposed by Boella and Lesmo [2]. The agent metaphor may be seen as an instance of Dennett's *intentional stance* [11] and is inspired by the interpretation of normative *multiagent* systems as dynamic social orders. According to Castelfranchi [8], a social order is a pattern of interactions among interfering agents "such that it allows the satisfaction of the interests of some agent". These interests can be a shared goal, a value that is good for everybody or for most of the members. But the agents attribute to the normative system, besides goals, also the ability to autonomously enforce the conformity of the agents to the norms by means of sanctions. In this approach the obligations of the agents can be formalized as desires or goals of the normative agent. This representation may be paraphrased as "Your wish is my command" because the desires or wishes of the normative agent are the obligations or commands of the other agents.

Most formalizations of normative systems, however, including [3], identify norms with obligations, prohibitions and permissions. This is not sufficient in complex normative multiagent systems for the following three reasons. First of all, these norms, called regulative norms, specify all the conditions when they are applicable. It would be more economic that regulative norms could factor out particular cases and could refer to

more abstract concepts only. Hence, the normative system should include mechanisms to introduce new legal categories of abstract entities for classifying possible states of affairs. Second, the dynamics of the social order is due to the fact that the normative system evolves over time by introducing new norms and abrogating outdated ones. So the normative system itself must specify how it can be changed by introducing new regulative norms, new legal categories and by whom the changes can be done. Third, the dynamics of a normative system includes the possibility that not only new norms are introduced by the legislators, but also that ordinary agents create new obligations and permissions concerning specific agents. This feature is fundamental to preserve the autonomy of agents inside an organization. In particular, it allows modelling contracts which introduce new normative relations among agents, like the duty to pay a fee for a service.

We therefore introduce a formal framework for the construction of normative multi-agent systems, based on Searle's notion of the construction of social reality. Searle [21] argues that there is a distinction between two types of rules, a distinction which also holds for formal rules like those composing normative systems:

> Some rules regulate antecedently existing forms of behaviour. For example, the rules of polite table behaviour regulate eating, but eating exists independently of these rules. Some rules, on the other hand, do not merely regulate an antecedently existing activity called playing chess; they, as it were, create the possibility of or define that activity. The activity of playing chess is constituted by action in accordance with these rules. The institutions of marriage, money, and promising are like the institutions of baseball and chess in that they are systems of such constitutive rules or conventions ([21], p. 131).

For Searle, institutional facts like marriage, money and private property emerge from an independent ontology of "brute" physical facts through constitutive rules of the form "such and such an X counts as Y in context C" where X is any object satisfying certain conditions and Y is a label that qualifies X as being something of an entirely new sort. E.g., "X counts as a presiding official in a wedding ceremony", "this bit of paper counts as a five euro bill" and "this piece of land counts as somebody's private property".

Like we formalize obligations in terms of desires and goals, in the next section, we formalize the constitutive rules as belief rules of the normative agent. E.g., consider a society which believes that a field fenced by an agent counts as the fact that the field is property of that agent. The fence is a physical "brute" fact, while being a property is an institutional fact attributed to the beliefs of the normative system. Regulative norms which forbid trespassing refer to the abstract concept of property rather than to fenced fields. As the system evolves, new cases are added to the notion of property, without changing the regulative norms about property. E.g., if a field is inherited, then it is property of the heir.

Searle's analysis of constitutive rules has focused mainly on the attribution of a new functional status to entities, as in the examples above: marriages, money, property. Searle's idea is that constitutive rules "create the possibility or define that activity". We believe, however, that the role of constitutive rules is not limited to the creation of an

activity and the construction of new abstract legal categories. Constitutive norms specify both the creation of legal categories and the evolution of the system: the normative system itself specifies by means of constitutive rules (included in its belief rules) how its beliefs, desires and goals can be changed, who can change them, and the limits of the possible changes. In this way, complex normative systems achieve a legal regime that includes rules conferring legal powers on certain participants: an agent is turned on a "private legislator" (Hart, [15]): "he is made competent to determine the course of law within the sphere of his contracts, trusts, wills and other structures [...] which he is enabled to build". Agents become able to design "relatively independent *institutional legal orders* within the comprehensive legal orders" (Ruiter [20]).

The regime of a legal institution can be defined as the set of legal consequences that flow from the existence of the institution. However, the meaning of "legal consequences" differs from what is normally understood by the term. Usually, since obligations have a conditional nature, when the conditions of an obligation are satisfied, as a legal consequence the addressee of the obligation is categorically obliged to fulfill it. Legal institutions, like contracts, marriages and properties, refer to a different kind of legal consequences; e.g., the legal rule "in a marriage parents have the reciprocal obligation to take care of and support their children" is not a conditional rule: it expresses the fact that when a legal institution of marriage comes into existence (say between Amy and Bob) only then the obligation that the spouses (Amy and Bob) take care and support their children is created. The same happens with the legal institution of contracts: when a contract comes into existence it creates obligations for the parties, i.e., new regulative rules which the normative system considers as its own. E.g., the Italian Civil Code art. 1173 (sources of obligations) specifies that obligations are created by contracts and art. 1372 (efficacy of contracts) that a contract has the strength of law (a contract is an agreement among two or more parties to regulate a juridical relationship about valuables *ex* art. 1321).

Moreover, contracts as legal institutions bring with them not only new regulative rules, but also constitutive ones which create new institutional facts and also new obligations; in this way, it is possible to specify in a contract new procedures for the interaction between parties, for specifying the evolution of the contract and how new obligations are created later. As Dignum *et al.* [12] notice, a contract specifies the events that alter the status of the contract. It is necessary to specify an interaction structure which indicates the possibility of an agent and the consequences of its choices; the contract must specify how to proceed if a norm is violated and what the violator is expected to do; e.g., if some payment deadline is not respected, the agent is obliged to pay a double fee. Since we model contracts as legal institutions, we are now aware that this rule is not a conditional obligation: it is an obligation created by some event specified in the contract, in the same way as the contract itself can create obligations. This is possible because we consider a contract as a legal institution, i.e., a normative system inside the main normative system: as a normative system it specifies who has the power to introduce obligations.

3 The conceptual model

In order to provide a formalization of contracts as legal institutions in organizations we first delineate the conceptual model we adopt.

First of all, the structural concepts and their relations. We have to describe the different aspects of the world and the relationships among them. We therefore introduce a set of propositional variables X and we extend it to consider also negative states of affairs: $L(X) = X \cup \{\neg x \mid x \in X\}$. Moreover, for $x \in X$ we write $\sim x$ for $\neg x$ and $\sim(\neg x)$ for x. The relations between the propositional variables are given by means of conditional rules written as $R(X) = 2^{L(X)} \times L(X)$: the set of pairs of a set of literals built from X and a literal built from X, written as $l_1 \wedge \ldots \wedge l_n \rightarrow l$, and, when $n = 0$, $\top \rightarrow l$. The rules will be used to represent the relations among propositional variables existing in beliefs, desires and goal of the agents.

Then there are the different sorts of agents A we consider. Besides real agents RA (either human or artificial) we consider as agents in the model also socially constructed agents like groups, normative systems and organizations SA. These latter agents do not exist in the usual sense of the term. Rather, they exist only as they are attributed mental attitudes by other agents (either real or not). By mental attitudes we mean beliefs B, desires D and goals G.

Coming to the relations existing between these structural concepts, mental attitudes, even if they do not coincide with, are represented by rules: $MD : B \cup D \cup G \rightarrow R(X)$. When there is no risk of confusion we will abuse the notation by identifying rules and mental states. To resolve conflicts among motivations we introduce a priority relation by means of $\geq$: $A \rightarrow 2^M \times 2^M$ a function from agents to a transitive and reflexive relation on the powerset of the motivations containing at least the subset relation. We write $\geq_a$ for $\geq(a)$. Moreover, different mental attitudes are attributed to all the different sorts of agents by the agent description relation $AD : A \rightarrow 2^{B \cup D \cup G \cup A}$. We write $B_a = AD(a) \cap B$, $A_a = AD(a) \cap A$ for $a \in A$, *etc.*

Also agents are in the target of the AD relation for the following reason: groups, normative systems, and organizations agents exist only as profiles attributed by other agents. So groups, normative systems and organizations exist only as they are described as agents by other agents, according to the agent description relation. The AD relation induces an exists-in-profile relation specifying that an agent $b \in SA$ exists only as some other agents attribute to it mental attitudes: $\{a \mid b \in A_a\}$.

Finally, the different sorts of agents are disjoint and are all subsets of the set of agents A: $RA \cup SA \subseteq A$.

We introduce now concepts concerning informational aspects. First of all, the set of variables whose truth value is determined by an agent (decision variables) [17] are distinguished from those which are not P (the parameters).

Besides, we need to represent also the so called "institutional facts" I. They are states of affairs which exist only inside normative systems and organizations. As discussed in the previous section, Searle [22] suggests, money, properties, marriages exist only as part of social reality; since we model social reality by means of the attribution of mental attitudes to social entities, institutional facts are just in the beliefs of these agents. Similarly, we need to represent that social entities like normative systems and organizations are able to change themselves. The actions determining the changes are

called creation actions C. Finally, we introduce contracts CT: they are agreements between agents in normative systems or organizations which have legal consequences; they are defined in Section 3.2.

As concerns the relations among these concepts, we have that parameters P are a subset of the propositional variables X. The complement of X and P represents the decision variables controlled by the different agents. Hence we have to associate to each agent a subset of $X \setminus P$ by extending again the agent description relation $AD : A \rightarrow 2^{B \cup D \cup G \cup A \cup (X \setminus P)}$.

Moreover, the institutional facts I are a subset of the parameters P: $I \subseteq P$. And the creation actions C are a subset of the institutional facts $C \subseteq I$: they do not exist outside the mind of agents and they have effects on the mental attitudes of agents only as far as the agents believe they have.

Since social entities depend on the attribution of mental attitudes, we represent their modification by means of the modification of their mental attitudes expressed as rules. So the creation action relation $CR : \{b, d, g\} \times A \times R(X) \rightarrow C$ is a mapping from rules (for beliefs, desires and goals) to propositional variables, where $CR(b, a, r)$ stands for the creation of $m \in B_a$, $CR(d, a, r)$ stands for the creation of $m \in D_a$, and $CR(g, a, r)$ stands for the creation of $m \in G_a$, such that the mental attitude is described by the rule $r \in R(X)$: $r = MD(m)$. For space reasons, in this paper we consider only the creation of new rules and not their deletion from the mental attitudes of an agent.

Since institutional facts I and the creation actions C exist only in the beliefs of a normative system or an organization, we need a way to express how these beliefs can be made true. As we discussed in the previous section, the relations among propositional variables are expressed as rules. In this case we have rules concerning beliefs about institutional facts: they are called constitutive rules and represent the "counts as" relations introduced by Searle [22] (see previous section). We thus identify the subset CN of the belief rules which express the relation between propositional variables and institutional facts: rules $C \cup \{x\} \rightarrow y \in R(X)$ expressing the fact that a literal $x \in L(X)$ in context $C \subseteq Lit(X)$ counts as the institutional fact $y \in L(I)$.

Finally, we have to model the effect of the creation actions on the mental attitudes of agents. For this reason we introduce an update relation UP from creation actions and mental attitudes to set of rules representing the new mental attitudes $UP : \{B, D, G\} \times C \rightarrow 2^{\{B,D,G\}}$. Since a decision of an agent can make true some creation actions, the consequences of these actions must be considered in the subsequent reasoning of the agent. The update function can be used to define the history of the multiagent system representing its evolution. Note that in this paper we do not consider the problem of the belief (and goal) revision. While this problem is sometimes addressed when dealing with normative systems, we consider here only the problem of introducing we rules and not of deciding which rules are necessary to get a certain revision.

We can now define a multiagent system as $MAS = \langle RA, SA, X, P, B, D, G, C, AD, MD, \geq, I, CT \rangle$.

We need to introduce normative multiagent systems to model organizations which are able to issue and enforce obligations: let the normative agent $\mathbf{o} \in SA$ be an agent representing the organization. Let the norms $\{n_1, \ldots, n_m\} = N$ be a set. Let the norm description $V : N \rightarrow X_{\mathbf{o}} \cup P$ be a complete function from the norms to the decision

variables of the normative agent together with the parameters: we write $V(n, a)$ for the decision variable which represents that there is a violation of norm n by agent $a \in A$. With these elements we define sanction based obligations in the next section. The tuple $\langle RA, SA, X, P, B, D, G, C, AD, MD, \geq, I, CT, \mathbf{o}, N, V \rangle$ is a normative multiagent system $NMAS$.

As concerns the behavior of agents, in Section 4, we introduce the games that can be played between two agents **a** and **o**. Before games, we have to introduce two further notions: consequences of beliefs and decisions of agents.

To incorporate the consequences of belief rules, we introduce a simple logic of rules called out: it takes the transitive closure of a set of rules, called reusable input/output logic in [18]; $out(E, S)$ be the closure of $S \subseteq L(X)$ under the rules E:

- $out^0(E, S) = S$
- $out^{i+1}(E, S) = out^i(E, S) \cup \{l \mid L \rightarrow l \in E, L \subseteq out^i(E, S))$ for $i \geq 0$
- $out(E, S) = \cup_0^\infty(E, S)$

We can now introduce decisions of agents; they must be consistent with the consequences of beliefs according to the two agents **a** ($out(B_\mathbf{a}, \delta)$) and **o** ($out(B_\mathbf{o}, \delta_\mathbf{o} \cup out(B_\mathbf{o}, \delta_\mathbf{a}))$). The set of decisions Δ is the set of subsets $\delta = \delta_\mathbf{a} \cup \delta_\mathbf{o} \subseteq L(X)$ such that their closures under the beliefs $out(B_\mathbf{a}, \delta)$ and $out(B_\mathbf{o}, \delta_\mathbf{o} \cup out(B_\mathbf{o}, \delta_\mathbf{a}))$ do not contain a variable and its negation.

3.1 Obligations

Since contracts affect the obligations of an agent, we must first summarize their definition given in [3]. Obligations are defined in terms of goals of the addressee of the norm **a** and of the organization **o**. The definition of obligation contains several clauses. The first one is the central clause of our definition and defines obligations of agents as goals of the normative agent, following the ‘Your wish is my command’ strategy [3]. The first clause says that the obligation is implied by the desires of agent **o**, implied by the goals of agent **o**.

The second and third clauses can be read as “the absence of p is considered as a violation”. The association of obligations with violations is inspired to Anderson [1]’s reduction of deontic logic to alethic logic. The third clause says that the agent desires that there are no violations.

The fourth and fifth clauses relate violations to sanctions. The fourth clause assumes that agent **o** is motivated not to count behavior as a violation and apply sanctions as long as their is no violation; otherwise the norm would have no effect. Finally, for the same reason, we assume in the last clause that the agent does not like the sanction.

Definition 1 (Obligation). *Let* $NMAS = \langle RA, SA, X, P, B, D, G, C, AD, MD, \geq, I, CT, \mathbf{o}, N, V\rangle$ *be a normative multiagent system.*

Agent $\mathbf{a} \in A$ *is obliged to decide to do* $x \in L(X_{\mathbf{a}} \cup P)$ *with sanction* $s \in L(X_{\mathbf{o}} \cup P)$ *if* $Y \subseteq L(X_{\mathbf{a}} \cup P)$ *in* $NMAS$*, written as* $NMAS \models O_{\mathbf{ao}}(x, s|Y)$*, if and only if:*

1. $Y \rightarrow x \in D_{\mathbf{o}} \cap G_{\mathbf{o}}$*: if agent* **o** *believes* Y *then it desires and has as a goal that* x*.*
2. $Y \cup \{\sim x\} \rightarrow V(\sim x, \mathbf{a}) \in D_{\mathbf{o}} \cap G_{\mathbf{o}}$*: if agent* **o** *believes* Y *and* $\sim x$*, then it has the goal and the desire* $V(\sim x, \mathbf{a})$*: to recognize it as a violation by agent* **a**.
3. $\top \rightarrow \neg V(\sim x, \mathbf{a}) \in D_{\mathbf{o}}$*: agent* **o** *desires that there are no violations.*
4. $Y \cup \{V(\sim x, \mathbf{a})\} \rightarrow s \in D_{\mathbf{o}} \cap G_{\mathbf{o}}$*: if agent* **o** *believes* Y *and decides* $V(\sim x, \mathbf{a})$*, then it desires and has as a goal that it sanctions agent* **a**.
5. $Y \rightarrow \sim s \in D_{\mathbf{o}}$*: if agent* **o** *believes* Y*, then it desires not to sanction* $\sim s$ *. This desire of the normative system expresses that it only sanctions in case of violation.*
6. $Y \rightarrow \sim s \in D_{\mathbf{a}}$*: if agent* **a** *believes* Y*, then it desires* $\sim s$*, which expresses that it does not like to be sanctioned.*

Permissions and prohibitions can be defined in terms of motivational attitudes, too [4].

As discussed in [3], sanctions or rewards are not the only possible motivations to stick to obligations, but they are necessary to cope for the worst cases.

3.2 Contracts

Contracts are part of the beliefs attributed to the organization **o**: this fact makes it possible that they change the beliefs of the organization according to what specified by the organization itself.

A contract $ct \in CT$ is created (a fact represented by the institutional fact $c \in I$) only if the organization believes that some other fact has as a consequence that c is true. More precisely, if there is some fact which counts as c for the organization **o**. This fact can be a brute fact in the world or another institutional fact. E.g., since contracts are created by agreements, the contract c is created by the signatures of two agents, two decision variables e and f: a constitutive norm in the belief rules of agent **o** ($e \wedge f \rightarrow c \in B_{\mathbf{o}}$). One reason why the creation of the contract c is introduced as an intermediary between the agreement and its legal effects is that, as many other institutional facts, it allows decoupling the conditions of the creation of the institutional facts from its legal effects. In this way, e.g., it is possible to specify new ways of creating the contract (for instance using an electronic signature) maintaining the same rules specifying its legal effects.

The effect a contract achieves is to modify the mental attitudes of the normative system. Usually, it adds more than one rule to the beliefs $B_{\mathbf{o}}$, the desires $D_{\mathbf{o}}$, or the goals $G_{\mathbf{o}}$ by making true some creation actions in C. Again, the creation actions are institutional facts: they are made true only if the organization **o** believes that they are made true by the creation of the contract: e.g., $c \rightarrow t \in B_{\mathbf{o}}$, is another constitutive rule, read as $c \in I$ counts as the creation action $t \in C$. Since a contract counts as several creation actions $t \in C$, the conclusion is that c works as an abstraction: rather than connecting the signatures of the agents with the creation actions, the contract unifies all the legal effects.

Finally, we consider which mental attitudes are changed. A contract modifies the mental attitudes of the normative system: since both regulative rules like obligations and constitutive ones like those composing contracts are themselves defined in terms of mental attitudes of the normative system, a contract can have legal effects.

By making true some creation actions, a contract is able to create regulative norms as the obligations of an agent **a** to pay ($pay \in X_{\mathbf{a}}$) in case the requested good has been shipped to him; $O_{\mathbf{ao}}(pay, s \mid shipped)$ is defined by the normative goal and desire that shipped goods are payed: $shipped \rightarrow pay \in D_{\mathbf{o}} \cap G_{\mathbf{o}}$; the goal and desire to consider the lack of payment for shipped goods as a violation: $shipped \wedge \neg pay \rightarrow V(\neg pay, \mathbf{a}) \in D_{\mathbf{o}} \cap G_{\mathbf{o}}$. And finally, the goal and desire to sanction violations: $V(\neg pay, \mathbf{a}) \rightarrow s \in D_{\mathbf{o}} \cap G_{\mathbf{o}}$; avoiding the sanction $\top \rightarrow \neg s$ is a desire of agents **a** and **o**, thus, it is a precondition of the obligation [5].

The creation of the contract achieves these effects on the mental attitudes of the organization **o** since it counts as a series of creation actions: that the goals and desires defining the obligation are added. Since the counts as relation is described by constitutive rules in the beliefs of agent **o** we have (as concerns goals):

$$
\begin{aligned}
\{ & \\
& c \rightarrow CR(g, \mathbf{o}, shipped \rightarrow pay), \\
& c \rightarrow CR(g, \mathbf{o}, shipped \wedge \neg pay \rightarrow V(\neg pay, \mathbf{a})), \\
& c \rightarrow CR(g, \mathbf{o}, V(\neg pay, \mathbf{a}) \rightarrow s) \\
\} & \subseteq B_{\mathbf{o}}.
\end{aligned}
$$

Also constitutive rules can be created by contracts: they are defined by belief rules of the normative system **o**, so they are created by a creation action $CR(b, \mathbf{o}, t) \in C$.

First of all, the contract may specify some institutional fact which should be used in the interaction. E.g., the shipment of the exchanged good is an institutional fact $shipped \in I$; the fact that the good has been shipped is not a brute fact of the world (the buyer cannot check it), rather it is a fact which holds if there is some document like the so called bill of lading ($bill \in P$) issued by a third party [14]: $bill \rightarrow shipped$ is the rule t added to the beliefs of the organization **o** by the creation action $CR(b, \mathbf{o}, t) \in C$; the creation action is a consequence of the contract c: the constitutive rule $c \rightarrow CR(b, \mathbf{o}, bill \rightarrow shipped) \in B_{\mathbf{o}}$ creates another constitutive rule.

Second, constitutive rules created by contracts will introduce new obligations and new constitutive rules. In this way a contract can specify how new obligations may arise during the interaction of the parties. We return on [12]'s example: if an agent does not pay the fee for a shipped good, it is obliged to pay a double sum of money ($pay2$): $O_{\mathbf{ao}}(pay2, s' \mid shipped \wedge \neg pay)$. This obligation is not a preexisting conditional obligation: it is created as a legal consequence of an event, the sanction s for not having payed the fee. The sanction s, in this case, rather than being a direct punishment for agent **a**, counts as the action of creating a second obligation.Note that this obligation

does not exist until the normative system recognizes a violation and applies the sanction s. This part of the contract is thus represented by the constitutive rules which create further constitutive rules about goals (where $s' \in X_{\mathbf{o}}$ is a sanction both feared by agent **a** and not desired by agent **o**): e.g.,

$$\{$$
$$c \rightarrow CR(b, \mathbf{o}, s \rightarrow CR(g, \mathbf{o}, \mathit{shipped} \wedge \neg pay \rightarrow pay2)),$$
$$c \rightarrow CR(b, \mathbf{o}, s \rightarrow CR(g, \mathbf{o}, \neg pay2 \rightarrow V(\neg pay2, \mathbf{a})))$$
$$\} \subseteq B_{\mathbf{o}}$$

In summary, a contract is defined as:

Definition 2 (Contract). *A contract $ct \in CT$ is a triple:*

1. *An institutional fact $c \in I$ representing that the contract $ct \in CT$ has been created.*
2. *A constitutive rule which makes true the institutional fact c: $Y \rightarrow c \in B_{\mathbf{o}}$ where $Y \subseteq L(X)$*
3. *A set of constitutive rules having as antecedent the creation c of the contract ct and as consequent creation actions modifying the mental attitudes of the organization* **o***: $c \rightarrow CR(E, \mathbf{o}, r) \in B_{\mathbf{o}}$ where $E = \{b, d, g\}$.*

4 Games

The advantage of the attribution of mental attitudes to organizations is that standard techniques developed in decision and game theory can be applied to reasoning on contracts. Here we consider a simple form of games of two stages only where an agent **a** takes the normative agent **o** into account by playing games with it.

When agent **a** takes its decision $\delta_{\mathbf{a}}$ it has to minimize its unfulfilled motivational attitudes. But when it considers these attitudes, it must not only consider its decision $\delta_{\mathbf{a}}$ and the consequences of this decision; it must consider also the decision $\delta_{\mathbf{o}}$ of the organization **o** and its consequences, for example that it is sanctioned by agent **o**. So agent **a** recursively considers which decision agent **o** will take depending on its different decisions $\delta_{\mathbf{a}}$. Note that here we assume that **o** is aware of agent **a**'s decision: hence, agent **o** takes its decision considering the legal effects of agent **a**'s decision on its beliefs and motivations using the update function UP and the creation actions made true by the decision $\delta_{\mathbf{a}}$.

Given a decision $\delta_{\mathbf{a}}$, a decision $\delta_{\mathbf{o}}$ is optimal for agent **o** if it minimizes the unfulfilled motivational attitudes in $D_{\mathbf{o}}$ and $G_{\mathbf{o}}$ according to the $\geq_{\mathbf{o}}$ relation. The decision of agent **a** is more complex: for each decision $\delta_{\mathbf{a}}$ it must consider which is the optimal decision $\delta_{\mathbf{o}}$ for agent **o**.

Definition 3 (Recursive modelling). *Let:*

- *the unfulfilled motivations of decision δ according to agent $\mathbf{a} \in A$ be the set of motivations whose body is part of the closure of the decision under the belief rules but whose head is not.*
  $U(\delta, \mathbf{a}) = \{m \in M \mid MD(m) = l_1 \wedge \ldots \wedge l_n \rightarrow l, \{l_1, \ldots, l_n\} \subseteq out(B_{\mathbf{a}}, \delta)$ *and* $l \notin out(B_{\mathbf{a}}, \delta)\}$.

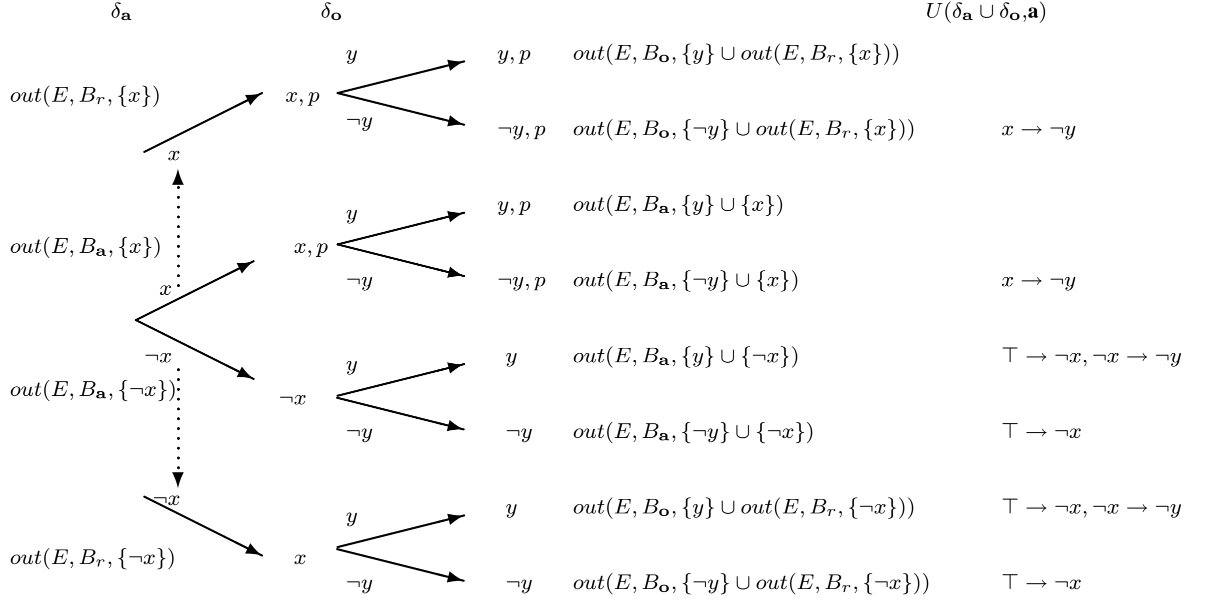


Fig. 1. The game between agent **a** and agent **o**.

- *the unfulfilled motivations of decision δ according to agent* **o** *be the set of motivations whose body is part of the closure of the decision under the belief rules and whose head is not, but considering the consequences of agent* **a***'s decision on agent* **o***'s beliefs and motivations. We write $O = out(B_{\mathbf{o}}, \delta_{\mathbf{a}}) \cap C$ for the set of creation actions which follow from $\delta_{\mathbf{a}}$:*

  $U(\delta, \mathbf{o}) = \{m \in UP(D_{\mathbf{o}} \cup G_{\mathbf{o}}, O) \mid MD(m) = l_1 \wedge \ldots \wedge l_n \rightarrow l, \{l_1, \ldots, l_n\} \subseteq out(UP(B_{\mathbf{o}}, O), \delta_{\mathbf{o}} \cup out(B_{\mathbf{o}}, \delta_{\mathbf{a}}))$ *and* $l \notin out(UP(B_{\mathbf{o}}, O), \delta_{\mathbf{o}} \cup out(B_{\mathbf{o}}, \delta_{\mathbf{a}}))\}$.

- *A decision δ (where $\delta = \delta_{\mathbf{a}} \cup \delta_{\mathbf{o}}$) is* optimal *for agent* **o** *if and only if there is no decision $\delta'_{\mathbf{o}}$ such that $U(\delta, \mathbf{o}) >_{\mathbf{o}} U(\delta_{\mathbf{a}} \cup \delta'_{\mathbf{o}}, \mathbf{o})$. A decision δ is optimal for agent* **a** *and agent* **o** *if and only if it is optimal for agent* **o** *and there is no decision $\delta'_{\mathbf{a}}$ such that for all decisions $\delta' = \delta'_{\mathbf{a}} \cup \delta'_{\mathbf{o}}$ and $\delta_{\mathbf{a}} \cup \delta''_{\mathbf{o}}$ optimal for agent* **o** *we have that $U(\delta', \mathbf{a}) >_{\mathbf{a}} U(\delta_{\mathbf{a}} \cup \delta''_{\mathbf{o}}, \mathbf{a})$.*

4.1 Example

We now return to the example about trade contracts. For space reasons, we formalize it as concerns only the obligation $O_{\mathbf{ao}}(pay, s \mid shipped)$ and the constitutive rule saying that the bill of lading counts as the good has been shipped.

We have two agents: the agent $\mathbf{a} \in RA$ and the organization $\mathbf{o} \in SA$. Agent **a** attributes mental attitudes to the organization **o** ($\mathbf{o} \in A_{\mathbf{a}}$).

The agent **a** can perform the actions of signing a contract and paying ($\{sign, pay\} \subseteq X_{\mathbf{a}}$), it believes that it has already signed the contract and the bill of lading $bill \in P$

has been issued $\{\top \rightarrow sign, \top \rightarrow bill\} \subseteq B_{\mathbf{a}}$, it desires not to give its money away ($\top \rightarrow \neg pay, \in D_{\mathbf{a}}$) and not to be sanctioned by agent **o** ($\top \rightarrow \neg s \in D_{\mathbf{a}}$).

The organization **o** does not desire to consider a violator ($V(\neg pay, \mathbf{a}) \in X_{\mathbf{o}}$) and to sanction agent **a** ($s \in X_o$) without motivation: $\{\top \rightarrow \neg V(\neg pay, \mathbf{a}), \top \rightarrow \neg s\} \subseteq D_{\mathbf{o}}$. It believes that if agent **a** signs ($sign$) the contract, this counts as the creation ($c \in I$) of the contract ($ct \in CT$): $sign \rightarrow c \in B_{\mathbf{o}}$. It believes that the contract has been signed and the bill of lading ($bill \in P$) has been issued $\{\top \rightarrow sign, \top \rightarrow bill\} \subseteq B_{\mathbf{o}}$ (as agent **a** does) and also the constitutive norms concerning the effects of the contract.

The first effect is that the new obligation to pay when the good is shipped is introduced: $O_{\mathbf{ao}}(pay, s \mid shipped)$. The obligation is defined by a set of desires and goals: the normative goal and desire that shipped goods are payed: $shipped \rightarrow pay \in D_{\mathbf{o}} \cap G_{\mathbf{o}}$; the goal and desire to consider the lack of payment for shipped goods as a violation: $shipped \wedge \neg pay \rightarrow V(\neg pay, \mathbf{a}) \in D_{\mathbf{o}} \cap G_{\mathbf{o}}$. And the goal and desire to sanction violations: $V(\neg pay, \mathbf{a}) \rightarrow s \in D_{\mathbf{o}} \cap G_{\mathbf{o}}$; note that the desire $\top \rightarrow \neg s$ of agents **a** and **o** are requested by the definition of obligation. The contract achieves these effects on the mental attitudes of the organization **o** since it counts as a series of creation actions: that the goals and desires defining the obligation are added. Since the counts as relation is described by constitutive norms, i.e., belief rules of agent **o**, we have:

$$
\begin{aligned}
\{ & \\
& c \rightarrow CR(g, \mathbf{o}, shipped \rightarrow pay), \\
& c \rightarrow CR(d, \mathbf{o}, shipped \rightarrow pay), \\
& c \rightarrow CR(g, \mathbf{o}, shipped \wedge \neg pay \rightarrow V(\neg pay, \mathbf{a})), \\
& c \rightarrow CR(g, \mathbf{o}, V(\neg pay, \mathbf{a}) \rightarrow s), \\
& c \rightarrow CR(d, \mathbf{o}, shipped \wedge \neg pay \rightarrow V(\neg pay, \mathbf{a})), \\
& c \rightarrow CR(d, \mathbf{o}, V(\neg pay, \mathbf{a}) \rightarrow s) \\
\} & \subseteq B_{\mathbf{o}}
\end{aligned}
$$

The second effect is that the bill of lading ($bill$) is considered as the proof that the good has been shipped; the contract creates a constitutive rule in the beliefs of the normative system **o**: $c \rightarrow CR(b, \mathbf{o}, bill \rightarrow shipped) \in B_{\mathbf{o}}$.

We adopt the perspective of agent **a** who has to decide whether to pay its fee or not. To take a decision agent **a** must recursively model the organization **o**'s decision. Agent **a** takes the decision whose consequences minimize its unfulfilled motivational attitudes given the decision of the organization and its consequences. Moreover, the decision of the organization **o** is assumed to be taken from the point of view which considers the legal effects in the consequences $out(B_{\mathbf{o}}, \delta_{\mathbf{a}})$ of agent **a**'s decision. Agent **a** has already signed the contract, so its signature counts as the creation of the contract with its legal effects, $c \in out(B_{\mathbf{o}}, \delta_{\mathbf{a}})$:

$O = out(B_{\mathbf{o}}, \delta_{\mathbf{a}}) \cap C =$
$\{$
$CR(g, \mathbf{o}, \mathit{shipped} \rightarrow \mathit{pay}),$
$CR(d, \mathbf{o}, \mathit{shipped} \rightarrow \mathit{pay}),$
$CR(g, \mathbf{o}, \mathit{shipped} \wedge \neg \mathit{pay} \rightarrow V(\neg \mathit{pay}, \mathbf{a})),$
$CR(g, \mathbf{o}, V(\neg \mathit{pay}, \mathbf{a}) \rightarrow s),$
$CR(d, \mathbf{o}, \mathit{shipped} \wedge \neg \mathit{pay} \rightarrow V(\neg \mathit{pay}, \mathbf{a})),$
$CR(d, \mathbf{o}, V(\neg \mathit{pay}, \mathbf{a}) \rightarrow s),$
$CR(b, \mathbf{o}, \mathit{bill} \rightarrow \mathit{shipped})$
$\}$

The updated beliefs and motivations are:

$UP(B_{\mathbf{o}}, O) \setminus B_{\mathbf{o}} = \{\mathit{bill} \rightarrow \mathit{shipped}\}$

$UP(D_{\mathbf{o}}, O) \setminus D_{\mathbf{o}} = UP(G_{\mathbf{o}}, O) \setminus G_{\mathbf{o}} = \{\mathit{shipped} \rightarrow \mathit{pay}, \mathit{shipped} \wedge \neg \mathit{pay} \rightarrow V(\neg \mathit{pay}, \mathbf{a}), V(\neg \mathit{pay}, \mathbf{a}) \rightarrow s\}$

The organization **o** has to decide whether agent **a**'s behavior respects the obligation or not; in the latter case agent **o** considers this as a violation and sanctions it.

The creation of the new constitutive rule has a further consequence, that the good has been shipped since the bill of lading counts as such:

$\mathit{shipped} \in out(UP(B_{\mathbf{o}}, O), \{\neg V(\neg \mathit{pay}, \mathbf{a}), \neg s\} \cup out(B_{\mathbf{a}}, \{\neg \mathit{pay}\})$

Thus, the new obligation $O_{\mathbf{ao}}(\mathit{pay}, s \mid \mathit{shipped})$ has its condition satisfied. If the agent decides not pay it violates its duty. Agent **o**'s unfulfilled mental attitudes are:

$U(\{\neg V(\neg \mathit{pay}, \mathbf{a}), \neg s\} \cup \{\neg \mathit{pay}\}, \mathbf{o}) \cap (D_{\mathbf{o}} \cup G_{\mathbf{o}}) =$
$\{\mathit{shipped} \rightarrow \mathit{pay}, \mathit{shipped} \wedge \neg \mathit{pay} \rightarrow V(\neg \mathit{pay}, \mathbf{a})\}$

We assume that fulfilling the set of motivations $\{\mathit{shipped} \rightarrow \mathit{pay}, \mathit{shipped} \wedge \neg \mathit{pay} \rightarrow V(\neg \mathit{pay}, \mathbf{a})\}$ is preferred, according to the ordering $\geq_{\mathbf{o}}$ on motivations, with respect to fulfilling $\{\mathit{shipped} \rightarrow \mathit{pay}, \top \rightarrow \neg V(\neg \mathit{pay}, \mathbf{a}), \top \rightarrow \neg s\}$: sanctioning violations worths its cost.

So the optimal decision for the organization is to consider **a**'s behavior as a violation and to sanction it $\delta_{\mathbf{o}} = \{V(\neg \mathit{pay}, \mathbf{a}), s\}$, as the unfulfilled motivations are:

$U(\{V(\neg \mathit{pay}, \mathbf{a}), s\} \cup \{\neg \mathit{pay}\}, \mathbf{o}) \cap (D_{\mathbf{o}} \cup G_{\mathbf{o}}) =$
$\{\mathit{shipped} \rightarrow \mathit{pay}, \top \rightarrow \neg V(\neg \mathit{pay}, \mathbf{a}), \top \rightarrow \neg s\}$

Instead, given the decision to pay the fee $\delta_{\mathbf{a}} = \{\mathit{pay}\}$, the optimal decision of agent **o** is not to consider as a violation the behavior of agent **a** and not to sanction it. Given $\delta_{\mathbf{o}} = \{\neg V(\neg \mathit{pay}, \mathbf{a}), \neg s\}$ the unfulfilled mental attitudes are:

$U(\{\neg V(\neg \mathit{pay}, \mathbf{a}), \neg s\} \cup \{\mathit{pay}\}, \mathbf{o}) \cap (D_{\mathbf{o}} \cup G_{\mathbf{o}}) = \emptyset$

How does agent **a** take a decision?

- if $\delta_{\mathbf{a}} = \{\neg \mathit{pay}\}$, then $\delta_{\mathbf{o}} = \{V(\neg \mathit{pay}, \mathbf{a}), s\}$:
  $U(\{V(\neg \mathit{pay}, \mathbf{a}), s\} \cup \{\neg \mathit{pay}\}, \mathbf{a}) \cap (D_{\mathbf{a}} \cup G_{\mathbf{a}}) = \{\top \rightarrow \neg s\}$
- if $\delta_{\mathbf{a}} = \{\mathit{pay}\}$, then $\delta_{\mathbf{o}} = \{\neg V(\neg \mathit{pay}, \mathbf{a}), \neg s\}$:
  $U(\{\neg V(\neg \mathit{pay}, \mathbf{a}), \neg s\} \cup \{\mathit{pay}\}, \mathbf{a}) \cap (D_{\mathbf{a}} \cup G_{\mathbf{a}}) = \{\top \rightarrow \neg \mathit{pay}\}$

If paying is preferred to being sanctioned $\{\top \rightarrow \neg s\} >_{\mathbf{a}} \{\top \rightarrow \neg \mathit{pay}\}$, agent **a** decides for $\delta_{\mathbf{a}} = \{\neg \mathit{pay}\}$.

5 Related work and summary

In this paper we address the problem of defining contracts as legal institutions. Using the methodology of attributing mental attitudes to social entities like organizations, we show that contracts have as precondition an agreement which counts as the creation of the contract and as legal consequences the creation of new mental attitudes. These attitudes define new obligations as well as new constitutive rules. We also show that the new constitutive rules can be used to prescribe the subsequent behavior expected by the parties involved in the contract.

What distinguishes our approach from other models of counts as relations is that we can connect goals, and obligations defined as goals, to institutional facts inside the overall frame of the attribution of the status of agent to the normative system: institutional facts are beliefs of the normative agent as any other belief.

Teague and Sonenberg [23] discuss the impact on reputation of levelled commitment contracts, i.e., contracts where each party can decommit by paying a predeterminate penalty. While reputation is beyond the scope of this paper, our model of contracts can specify also the procedures for the parties' decommitment.

Dignum *et al.* [12] propose the language *LCR* for modelling contracts. They define contracts as tuples composed of agents, contract clauses, stages and interactional structure. With respect to their work we do not define the clauses of a contract as conditional obligations (as also Pacheco and Carmo [19] do). Rather we use constitutive rules which create obligations when the contract is created or when some relevant event happens. Finally, as they propose, we give a definition of obligations in terms of violations but we take a subjective perspective and consider the decision problem of an agent subject to obligations.

Daskalopulu and Maibaum [9] model contracts as processes having as states legal relations among the parties. They introduce obligations which are consequences of the unfulfillment of other obligations. However, they do not consider the role of constitutive rules in contracts and the fact that violations are recognized only as an effect of the activity of the normative agent.

Future work concerns extending the games with multiple stages, so to track the evolution of the contract. We also propose contracts for modelling roles using the agent metaphor. Contracts are used for assigning roles: they create the obligations of the holder of a role starting from the description of the role. Moreover, roles and functional areas in an organization can be created as new legal institutions. Finally, our approach can be used to model security issues in virtual communities of agents. Contractual access control has been proposed in recent developments to cope with the issue of delegating powers to modify rights and permissions.

References

1. A. R. Anderson. The logic of norms. *Logic et analyse*, 2, 1958.
2. G. Boella and L. Lesmo. A game theoretic approach to norms. *Cognitive Science Quarterly*, 2(3-4):492–512, 2002.
3. G. Boella and L. van der Torre. Attributing mental attitudes to normative systems. In *Procs. of AAMAS'03*, pages 942–943. ACM Press, 2003.

4. G. Boella and L. van der Torre. Local policies for the control of virtual communities. In *Procs. of IEEE/WIC Web Intelligence Conference*, pages 161–167. IEEE Press, 2003.
5. G. Boella and L. van der Torre. Permissions and obligations in hierarchical normative systems. In *Procs. of ICAIL'03*, pages 109–118, Edinburgh, 2003. ACM Press.
6. G. Boella and L. van der Torre. Contracts as legal institutions in organizations of autonomous agents. In *Procs. of AAMAS'04*, New York, 2004.
7. G. Boella and L. van der Torre. Regulative and constitutive norms in normative multiagent systems. In *Procs. of 9th International Conference on the Principles of Knowledge Representation and Reasoning*, Whistler (CA), 2004.
8. C. Castelfranchi. Engineering social order. In *Procs. of ESAW'00*, pages 1–18. Springer Verlag, 2000.
9. A. Daskalopulu and T.S.E. Maibaum. Towards electronic contract performance. In *Procs. of Legal Information Systems Applications*, pages 771–777, 2001.
10. C. Dellarocas. Negotiated shared context and social control in open multi-agent systems. In R. Conte and C. Dellarocas, editors, *Social Order in MAS*. Kluwer, 2001.
11. D. C. Dennett. *The Intentional Stance*. The MIT Press, Cambridge, MA, 1987.
12. V. Dignum, J-J. Meyer, and H. Weigand. Towards an organizational-oriented model for agent societies using contracts. In *Procs. of AAMAS'02*. ACM Press, 2002.
13. P. J. Gmytrasiewicz and E. H. Durfee. Formalization of recursive modeling. In *Procs. of first ICMAS'95*, 1995.
14. J. Gordijn and Y.-H. Tan. A design methodology for trust and value exchanges in business models. In *Procs. of 16th BLED Conference*, pages 423–432, 2003.
15. H. L. A. Hart. *The Concept of Law*. Clarendon Press, Oxford, 1961.
16. A. Jones and J. Carmo. Deontic logic and contrary-to-duties. In D. Gabbay, editor, *Handbook of Philosophical Logic*, pages 203–279. Kluwer, 2001.
17. J. Lang, L. van der Torre, and E. Weydert. Utilitarian desires. *Autonomous Agents and Multi-agent Systems*, pages 329–363, 2002.
18. D. Makinson and L. van der Torre. Input-output logics. *Journal of Philosophical Logic*, 29:383–408, 2000.
19. O. Pacheco and J. Carmo. A role based model of normative specification of organized collective agency and agents interaction. *Autonomous Agents and Multi-agent Systems*, 6:145–184, 2003.
20. D.W.P. Ruiter. A basic classification of legal institutions. *Ratio Juris*, 10(4):357–371, 1997.
21. J.R. Searle. *Speech Acts: an Essay in the Philosophy of Language*. Cambridge University Press, Cambridge, England, 1969.
22. J.R. Searle. *The Construction of Social Reality*. The Free Press, New York, 1995.
23. V. Teague and L. Sonenberg. Investigating commitment flexibility in multi-agent contracts. In *Procs. of decision theoretic and game theoretic agents workshop at ICMAS'00*, Boston, 2000.

Integrating tuProlog into DCaseLP to Engineer Heterogeneous Agent Systems*

Ivana Gungui and Viviana Mascardi

Dipartimento di Informatica e Scienze dell'Informazione – DISI,
Università di Genova, Via Dodecaneso 35, 16146, Genova, Italy.
1995s133@educ.disi.unige.it, mascardi@disi.unige.it

Abstract. This paper discusses the integration of a Prolog implementation, tuProlog, into the DCaseLP environment for building prototypes of multi-agent systems (MASs). DCaseLP aims at providing the MAS developer with a plethora of specification and implementation languages in order to allow him/her to adopt the best language for each view of the system under specification/implementation. The integration of tuProlog into DCaseLP represents a step forward in this direction and allows the re-use of tools and mechanisms previously developed for the DCaseLP predecessor, CaseLP.

1 Introduction

Multiagent Systems (MASs) involve heterogeneous components which have different ways of representing their knowledge about the world, about themselves, and about other agents, and which adopt different mechanisms for reasoning about this knowledge. Despite heterogeneity, agents need to interact and exchange information in order to cooperate or compete for the control of shared resources; this interaction may follow sophisticated communication protocols.

For these reasons and due to the complexity of agents behaviour, MASs are difficult to be correctly and efficiently engineered. Even developing a working prototype may require a long time and a lot of different skills. In fact, the prototype can involve agents that would be better modelled and implemented by means of a language based on Horn clauses, agents that would be easily defined using an expert system-like language, and others that should be directly implemented in some implementation language, in order to access existing software packages or the web. Moreover, some general aspects of the MAS can be better specified with ad-hoc specification languages. For example, the MAS architecture, the internal agent architecture and the interaction protocols among agents can be easily specified using graphical tools and languages.

The development of a prototype system of heterogeneous agents can be carried on in different ways. A first -trivial- solution consists of developing all the agents by means of the same implementation language and to execute the obtained program. If this approach is adopted, during the specification stage it would be natural to select a specification language that can be either directly executed or easily translated into code, and to specify all the agents in the MAS using it. An opposite solution would

* Parts of this document appear in [5].

be to specify each "view" of the MAS (including the MAS architecture, the interaction protocols among agents, the internal architecture and functioning of each agent) using the most suitable language capable to deal with the MAS's peculiar features, and to verify, execute, or animate the obtained specifications inside an integrated environment. Such an environment should offer the means to select the proper specification language for each view of the MAS, and to check the specifications. This check may be carried out thanks to formal validation and verification methods or by producing an executable code and running the prototype thus obtained.

Despite its greater complexity, the last solution has many advantages over the first, trivial one.

1. By allowing the use of different specification languages for each view of the MAS, *it supports the progressive refinement of specifications*: for example, the specification of an interaction protocol performed during the early analysis stage does not need to be as detailed as the complete specification of an agent performed during the design stage; details can be progressively added while the engineering process goes on.
2. By allowing the use of different specification languages for the internal architecture and functioning of each agent, *it respects the differences existing among agents*, namely the way they reason and the way they represent their knowledge, the other agents, and the world.
3. By allowing different implementation languages to be integrated inside the same running prototype, *it allows the direct implementation of some of the agents*, skipping the specification stage.
4. In case more than one language fits the requirements of an agent/view under specification, *it allows the developer to choose the language he/she knows best and likes*, thus leading to more reliable specifications and implementations.

Currently, solid and complete environments that allow the integration of heterogeneous specification and implementation languages in a seamless way do not exist yet, but some preliminary steps have been made in this direction, and some initial results have already been achieved with the development of prototypical environments for engineering heterogeneous agents. DCaseLP (Distributed CaseLP), integrates a set of specification and implementation languages in order to model and prototype MASs and defines a methodology which covers the engineering stages from requirements analysis to prototype execution, which relies on the use of AUML (Agent UML, [14]) both at the requirement analysis level and for describing the *interaction protocols* followed by the agents. Although the first release of DCaseLP [12,1] demonstrates that the concepts underlying the "integrated environment for engineering heterogeneous MAS" can be put into practice and can give interesting results, it suffers from two limitations that affect its applicability:

1. it does not provide the means to re-use the code and instruments already developed for the predecessor of DCaseLP, CaseLP [13]; and
2. it does not provide tools and languages for reasoning about properties of the interactions occurring among the agents.

The last limitation can be addressed by translating AUML interaction protocols into the DyLOG language [8,4,6], and then integrating DyLOG into DCaseLP. The exploitation of DyLOG to address the problems of protocol selection, composition and implementation conformance w.r.t. an AUML sequence diagram is dealt with in [7], while the integration of DyLOG into DCaseLP is discussed in [5].

The first limitation can be overcome by extending DCaseLP with the ability to integrate agents specified as Prolog theories, as shown in this paper.

The structure of the paper is the following: Section 2 overviews the DCaseLP environment and discusses the outcomes of integrating an existing Prolog implementation, tuProlog, into DCaseLP, while Section 3 discusses the technical details of this integration. Section 4 shows an example of use of DCaseLP extended by tuProlog; conclusions follow.

2 The DCaseLP environment

DCaseLP is a prototyping environment where agents specified and implemented in a given (and fairly limited!) set of languages can be seamlessly integrated. DCaseLP provides an agent-oriented software engineering methodology to guide the MAS developer during the analysis of the MAS requirements, the MAS design, and the development of a working MAS prototype. The methodology is shown in Figure 1. Solid arrows represent the information flow from one step to the next one. Dotted arrows represent the iterative refinement of previous choices. The first release of DCaseLP did not deal with all the stages of the methodology. In particular, the verification stage was not addressed. In the same way as the integration of DyLOG into DCaseLP will allow us to formally verify properties of communication protocols, the integration of tuProlog into DCaseLP discussed in Section 3 will allow us to address the verification phase by re-using the verification mechanisms developed for CaseLP ([13], Sections 3.3 and 4.4).

DCaseLP is the result of the effort to re-implement CaseLP [13] in order to overcome its main limitations, namely:

1. its centralization,
2. its poor support to concurrency, and
3. its lack of adherence to existing standards.

The tools and languages supported by the first release of DCaseLP, discussed in [12,1], are represented in Figure 2 by means of the darker boxes. Lighter boxes represent the desired extensions in respect to that release. Some of these extensions have already been made, while some are currently being made, and some others are just part of our future work.

DCaseLP adopts an existing multiview, use-case driven and UML-based method [2,3] in the phase of requirements analysis.

Once the requirements of the application have been clearly identified, the developer can use UML and its agent-oriented extension AUML to describe the interaction protocols followed by the agents, the general MAS architecture and the agent classes and instances. Moreover, the developer can also automatically create the rule-based code for the agents in the MAS in such a way that the UML/AUML specification is satisfied.

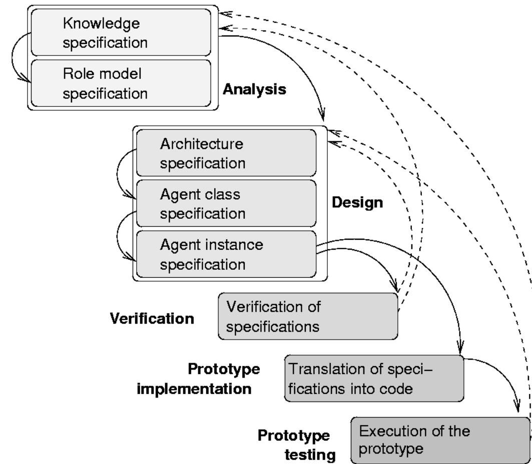


Fig. 1. DCaseLP methodology.

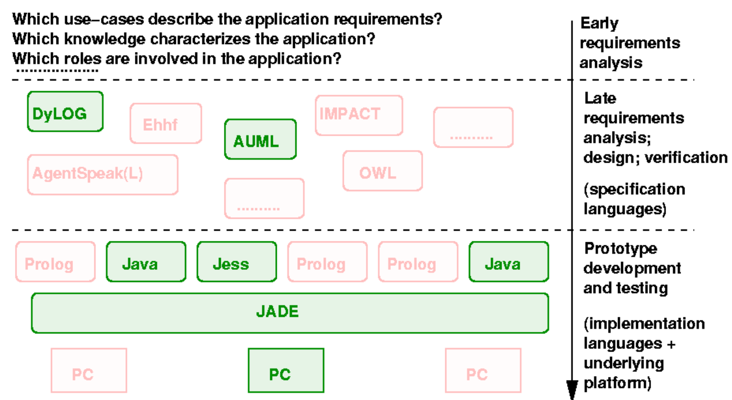


Fig. 2. Tools and languages supported by DCaseLP, first release.

In the following, we will assume to use AUML during the requirements analysis stage, although the translation from AUML into rule-based code is not fully automated (while the translation from pure UML into code is).

The rule-based language used for the implementation of DCaseLP agents is Jess [11]. The Jess code obtained from the translation of AUML diagrams must be manually completed by the developer with the behavioural knowledge which was not explicitly provided at the specification level. On the one hand, the developer does not need a deep insight in rule-based languages in order to complete the Jess code, since he/she is guided by comments included in the automatically generated code. In this way, a developer who is not confident with rule-based languages can concentrate on the AUML specification and make a little effort to complete the rule-based code in order to make it executable. On the other hand, the developer who prefers to define agents in a declarative language can skip the AUML specification stage and directly write the Jess code.

The choice of Jess as the language for implementing agents was lead by two considerations:

1. being a rule-based language, Jess is suitable for representing both the event-driven and the goal-driven behaviours of the agents;
2. being implemented in Java, Jess can be easily integrated into the FIPA-compliant JADE platform.

JADE (Java Agent Development Framework, [9]) provides both a middle-ware that complies with the FIPA specifications [10] and a set of graphical tools that support the debugging and deployment phases. The agents can be distributed across several machines and they can run concurrently. The adoption of JADE as the underlying platform for implementing DCaseLP was a must in order to overcome the three limitations of CaseLP. In fact, JADE is distributed, allows the concurrent execution of agents, and is FIPA-compliant. By integrating Jess into JADE, we were able to easily monitor and debug the execution of Jess agents thanks to the monitoring facilities that JADE provides. The experiments carried out with the first release of DCaseLP were on a single machine (see Figure 2: there is only one dark box labelled with “PC” under the JADE box).

The possibility of running the prototype allowed the first release of DCaseLP to demonstrate its ability in checking the coherence of the AUML diagrams produced during the requirements analysis step. Performing such a check is a well known and still open problem that we could face without additional effort. Nevertheless, that release still suffered from one limitation: it was not able to integrate any Prolog implementation. The predecessor of DCaseLP, namely CaseLP, is implemented in Sicstus Prolog [15], and a lot of work has been done to study and define semi-automatic translators from high-level specification languages into CaseLP agents, namely agents described in Sicstus Prolog extended with communication primitives. Limited support to formal verification of specifications – completely missing in DCaseLP – is indeed provided by CaseLP. Without the integration of Prolog into DCaseLP, all that work would have been lost. Recently, we have extended DCaseLP with the ability to integrate agents specified as Prolog theories. Section 3 discusses how we have integrated an existing Prolog implementation, tuProlog [16], into DCaseLP. The choice of tuProlog was due to two of its features:

1. it is implemented in Java, which makes its integration into JADE easier, and
2. it is very light, which ensures a certain level of efficiency to the prototype.

The integration of tuProlog into DCaseLP has been completed very recently. Due to the syntactic differences existing between Sicstus Prolog and tuProlog, CaseLP agents specified using Sicstus Prolog cannot be simply treated as if they were DCaseLP agents specified using tuProlog: a translation step from "Sicstus Prolog for CaseLP agents" to "tuProlog for DCaseLP agents" is necessary. We guess that this translation step can be easily automatised, thus allowing us to re-use the tools developed for CaseLP inside DCaseLP; however, its implementation has not been completed due to lack of time. Again due to time limitations, we did not verify the ability to run JADE, Jess and tuProlog agents as part of the same, heterogeneous, MAS. At the time of writing, we have only developed some examples (one of which is discussed in Section 4) that demonstrate that tuProlog agents are able to interact with both tuProlog and JADE agents by taking advantage of the underlying communication middle-ware provided by JADE, and that the execution of the resulting MAS can be monitored using the tools offered by JADE. When the translator "*Sicstus Prolog → tuProlog*" will be ready, and when the compatibility between Jess and tuProlog agents will be fully established, DCaseLP will be closer than now to the integrated environment for engineering heterogeneous MASs envisaged in Section 1. In particular,

1. *It will support the progressive refinement of specifications*: for example, the interactions among agents belonging to the MAS and among internal components of the same agent will be specified in some suitable language (AUML, other languages provided by CaseLP), will be then formally verified, and will be finally implemented by adding all the details needed by the MAS or by the single agent to work.
2. *It will respect the differences existing among agents*: an agent which reasons in a goal-driven, backward fashion will be easily defined by means of a tuProlog theory; a rule-based agent will be better defined using Jess.
3. *It will allow the direct implementation of some of the agents*: JADE agents are basically Java agents and thus they are implemented agents, rather than specified agents.
4. *It will allow the developer to choose the language he/she knows best and likes*: it will provide a bunch of languages to choose from.

3 Integrating tuProlog into DCaseLP

The integration of tuProlog into DCaseLP has been carried out in order to provide the developer of the MAS with a means to define the behaviour of an agent by using another declarative language besides Jess, and to re-use the code and instruments previously developed for CaseLP. To do so, tuProlog has been integrated into JADE.

JADE includes a specific package to develop Java agents and a programmer's guide containing implementation guidelines that the developer should follow to code his/her agents in Java. Any Java class that extends the class `Agent` defined in the package `jade.core` of JADE can be considered as a JADE agent. To add tuProlog in DCaseLP, three Java classes have been defined in a package named `tuPInJADE`:

1. the class JadeShell42P, which represents a tuProlog agent in JADE;
2. the class JadeShell42PGui that provides an additional GUI at the loading of the agent; and
3. the class TuJadeLibrary, which is a tuProlog library (developed in Java) necessary to a tuProlog agent in order to communicate in the JADE platform.

As the name of the class suggests, JadeShell42P behaves as a shell for a tuProlog engine. To execute a JadeShell42P agent in JADE, the programmer has to give, in input, the name of a file containing a tuProlog theory that represents the behaviour of the agent (Figure 3). The class JadeShell42PGui differs from class

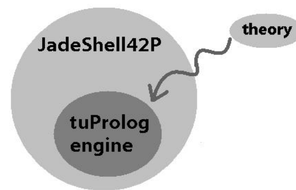


Fig. 3. JADE shell for a tuProlog engine.

JadeShell42P in the fact that, when loaded into JADE, it does not need the name of the theory file in the command line: it loads the pop-up window shown in Figure 4 with which the user can browse the file system and select from the list of files the one defining a tuProlog theory to be used as behaviour of the agent. Such a tuProlog the-

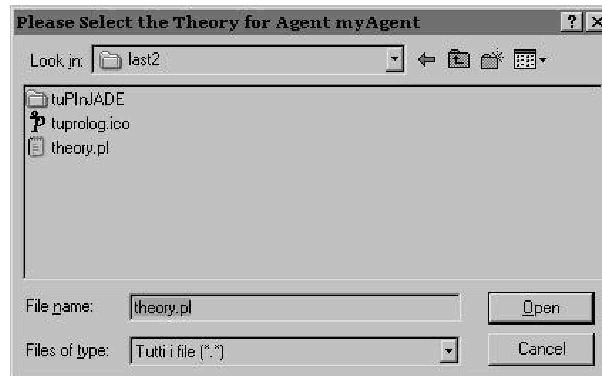


Fig. 4. Window for theory selection.

ory file has only one restriction: it has to begin with the definition of a predicate called

main/0. When a tuProlog agent is loaded into JADE, it first creates a tuProlog engine that supports the standard tuProlog libraries and then extends them by loading the ad-hoc tuProlog library named TuJadeLibrary. The behaviour of any tuProlog agent is to use the tuProlog engine, created during its initialisation phase, to always solve the predicate main. A typical main predicate calls predicates to read a new message, handle it and carry out some actions such as update of the agent's knowledge and message delivery.

The goal's demonstration is not visible to the programmer: if he/she wants to be informed of the variable's bindings made during resolution, he/she has to explicitly write the variables on the standard output or in some files that he/she can subsequently go to and read. The only explicit information which is provided for the user regards the failure of the goal's demonstration and other situations which raise an error during the resolution process. To make this information visible, the package tuPInJADE defines the Java class ErrorMsg, that is used by the tuProlog agents as a pop-up window displaying error and failure messages, like the one shown in Figure 5. The

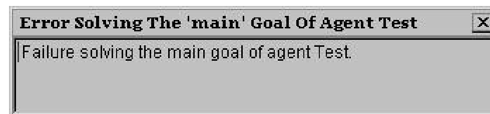


Fig. 5. Window for error and failure messages.

Java class JadeShell42P defines the inner class Shell42PBehaviour (named Shell42PBehaviourGui in the class JadeShell42PGui) that extends the Java class CyclicBehaviour defined in the package jade.core.behaviours of JADE.

Shell42PBehaviour implements the only behaviour of a JadeShell42P agent: every time the agent is scheduled by the JADE's scheduler, it tries to fulfill only one activity, that is, the resolution of the goal main. The Shell42PBehaviour models a cyclic task and cannot be interrupted while executing its action method. The result is the same as if the agent performed a "while true do main" statement, with main being dealt with as an atomic action.

The Java class TuJadeLibrary is the core class dealing with communication of tuProlog agents in JADE. This library defines the predicates send and receive: they are the directives implementing the sending and receiving of the FIPA compliant and asynchronous messages to and from agents of a JADE platform. The send and receive predicates simply invoke the send and receive methods of a JADE agent, therefore they not only allow communication among tuProlog agents but also among ordinary JADE agents and tuProlog agents.

The arguments of the send predicate are: the performative, the content and the JADE address/list of addresses of the receiver/receivers of the message. The arguments of the receive predicate are: the performative, the content and the JADE address of the sender of the message. Actually, since JADE agents have the possibility to stop their

activity while waiting for a message to arrive in their messages queue, the TuJadeLibrary also defines two blocking_receive predicates: one without a timeout and the other with a timeout. These predicates correspond to the blockingReceive method of an ordinary JADE agent.

Finally, TuJadeLibrary defines two predicates for converting strings into terms and vice-versa, named pack and unpack. They allow tuProlog agents to send strings as the content of their messages, and to reason over them as if they were tuProlog terms.

4 Example

To show how DCaseLP can be used to develop a working MAS prototype, we use a simple example drawn from a distributed marketplace scenario.

In such a marketplace, there are two agents (buyer1 and buyer2) that want to buy some fruit (oranges, apples and kiwi) from three agents (seller, seller1 and seller2). Agents buyer1, buyer2, seller1 and seller2 are all tuProlog agents, while seller is an ordinary JADE agent.

The agents that sell fruit can receive two kinds of FIPA ACL messages from the buyers:

1. a request for price: the message received has the performative REQUEST and the content price(Fruit), where Fruit is oranges or apples or kiwi;
2. a request for buying: the message received has the performative REQUEST and the content buy(Fruit, Amount), where Fruit is oranges or apples or kiwi, while Amount is the quantity of fruit that the buyer wants to buy.

A seller replies to a price request made by a buyer by sending an INFORM message that has the content price(Fruit, Price), where Fruit is oranges or apples or kiwi and Price is the corresponding price.

The reply to a request for buying depends on whether or not the seller has enough fruit to sell: in case the quantity of fruit that the buyer is willing to buy is less or equal to the one possessed by the seller, the seller will send the buyer an INFORM message with the content bought(Fruit), to inform the buyer that the fruit Fruit has been sold. On the other hand, if the seller does not own enough fruit, it sends the buyer an INFORM message with the content no_more(Fruit), so the buyer will know it can no longer buy Fruit from that seller.

At the beginning, the buyers send a request for the price of all the fruit to all the sellers. Once they know the prices of the fruit, they send requests for buying fruit to the agents that sell it at the cheapest price. The buyers keep sending messages requesting to buy fruit while they still have money and the sellers have enough fruit to sell.

To give the flavor of how a tuProlog agent looks like, the code below shows a piece of the tuProlog theory defining the behaviour of buyer1.

```
main :-
  handle_msgs,
  ask_prices,
  buy_goods.
```

```
goods_possessed(oranges, 0) :- true.
goods_possessed(apples, 0) :- true.
goods_possessed(kiwi, 0) :- true.

buys(goods(oranges), quantity(2)) :- true.
buys(goods(apples), quantity(3)) :- true.
buys(goods(kiwi), quantity(12)) :- true.

money(200) :- true.

sellers_addresses(["seller1@gruppoai:1099/JADE",
             "seller2@gruppoai:1099/JADE",
             "seller@gruppoai:1099/JADE"]) :- true.

.............

handle_msgs :-
  receive(Performative, Message, Sender),
  select(Performative, Message, Sender).

select(Performative, Message, Sender) :-
  bound(Performative),
  bound(Message),
  address_name(Sender, Name),
  unpack(Message,TermMsg),
  handle(Performative, TermMsg, Sender).

select(_, _, _) :- true.

handle("INFORM",
      bought(Goods),
      Sender) :-
  bound(Goods),
  address_name(Sender,S),
  price(S,Goods,P),
  retract(money(M)),
  retract(goods_possessed(Goods,X)),
  buys(goods(Goods),quantity(Q)),
  N is X + Q,
  P \= na,
  NM is M - P,
  assert(money(NM)),
  assert(goods_possessed(Goods,N)).
```

The main predicate defines three activities which consist in handling incoming messages, asking the price of fruit from sellers (only at the beginning, when the buyer does not yet know the prices) and buying fruit. After defining the main predicate, the theory declares the initial state of the buyer: buyer1 possesses no fruit, buys oranges in stocks of 2 kilograms, apples in stocks of 3 kilograms and kiwi in stocks of 12 kilograms, and has 200 Euro to spend. The list of addresses of the sellers follows (sellers_addresses), together with other information not relevant in this context.

The handling of messages consists of receiving one message (calling the receive predicate provided by the TuJadeLibrary and introduced in Section 3) and transforming its content, which is a string, into a tuProlog term (calling the user-defined predicate select). The select predicate calls the unpack predicate provided by the TuJadeLibrary in order to transform the string that represents the content of the message into a term, and then it calls the user-defined handle predicate on the performative of the message, the obtained term, and the sender of the message.

In the example considered, a buyer receives a message whose content is the string bought(Goods). The buyer knows the price of Goods (by solving the goal price(S, Goods,P)) and it knows the quantity of Goods it bought (by solving the goal buys(goods(Goods),quantity(Q))). Having succeeded in buying Goods, the buyer must update both the possessed amount of Goods and the remaining money (calls to standard Prolog predicates retract, is and assert). Similar definitions of the predicate handle are provided for any other message that the buyer may receive.

The ordinary JADE agent, seller, is characterised by a Java code partly shown below.

```
package tuPInJADE;

import jade.core.Agent;
import jade.core.AID;
import jade.core.behaviours.CyclicBehaviour;
import jade.lang.acl.ACLMessage;

public class Seller extends Agent
   { private int orangesAmount = 5;
    private int applesAmount = 5;
    private int kiwiAmount = 10;
    private int orangesPrice = 105;
    private int applesPrice = 80;
    private int kiwiPrice = 100;

  protected void setup()
        { SellBehaviour p = new SellBehaviour(this);
        addBehaviour(p);
        }}

class SellBehaviour extends CyclicBehaviour
```

```
{ private static boolean done = false;

public SellBehaviour(Agent a)
     { super(a); }

public void action()
     { ACLMessage msg;
     while (!done)
     { msg = myAgent.receive();
     if (msg != null) handleMsgs(msg);}}
```

The `Seller` class extends the JADE `Agent` class as any agent running in JADE must do. The behaviour of the seller is a cyclic behaviour (class `SellBehaviour` extends `CyclicBehaviour`) which continuously checks for a message (`msg = myAgent.receive()`) and, if a message is present, handles it (`if (msg != null) handleMsgs(msg)`).

Once all the agents have been specified using tuProlog or JADE, they can be loaded into JADE and the execution of the obtained prototype can start. JADE offers the possibility to follow the communication between the agents by means of the "sniffer" agent which is a GUI whose output is shown in Figure 6.

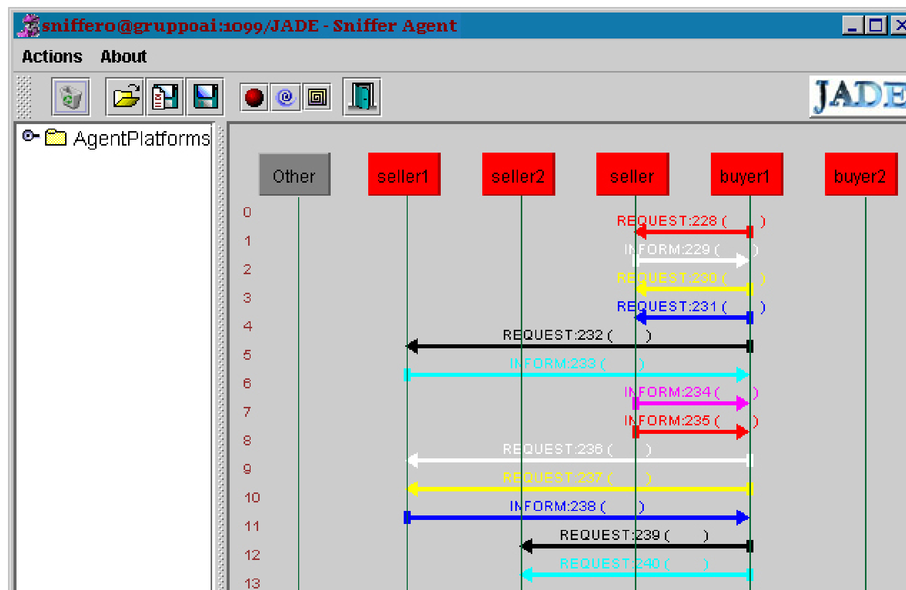


Fig. 6. Output of the JADE sniffer agent.

The state of the agents' mailboxes can be inspected thanks to the introspector agent, a GUI too. Figure 7 shows the state of the mailbox of `buyer2`. This screen-shot was taken at the beginning of the simulation; all the `INFORM` messages shown are answers to price requests previously issued by `buyer2` to the sellers.

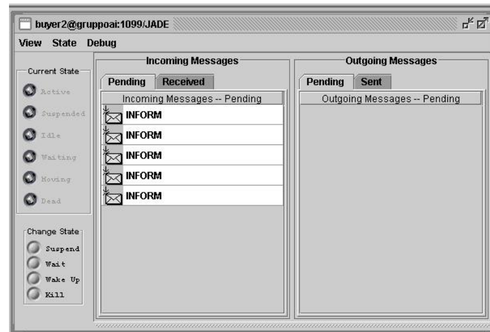


Fig. 7. JADE window showing the communication among agents.

Details on the messages exchanged can also be inspected. Figure 8 shows the request for the price of kiwi sent by `buyer2` to `seller1`. Figure 9 shows the answer to this request.

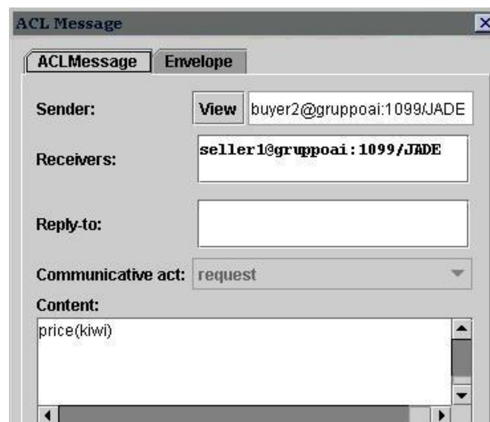


Fig. 8. Price request from `buyer2` to `seller1`.

The execution and monitoring of the prototype, obtained by exploiting the tools provided by JADE, allow the developer to verify whether the agents work well according

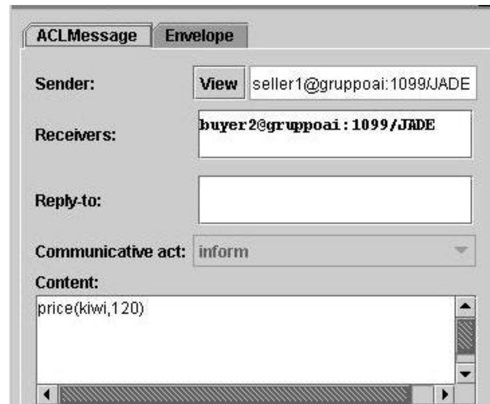


Fig. 9. Price answer from `seller1` to `buyer2`.

to their intended behaviour. The sniffer agent also allows to save into a file the messages exchanged by the agents. When that file is loaded by the user through the sniffer agent, it is possible to view the details of each message by clicking on the arrow representing the exchange of a message. A user can then check if the messages have been sent in the expected order (for example, that all the buyers ask for the price of fruit first, and start buying fruit afterwards), by viewing the content of every single message displayed in the canvas of the agent sniffer. Without the integration of tuProlog into JADE, verifying the correctness of communication between agents implemented in Prolog could only be done by hand: the developer had to put breakpoints in his/her code or he/she had to write messages on the standard output or in a separate file in order to follow what was going on during the prototype execution. CaseLP offers graphical debugging tools more sophisticated than this "by-hand" inspection. Nevertheless, the adoption of the instruments already provided by a standard, FIPA-compliant and open-source platform, represents an improvement to the use of proprietary instruments offered by CaseLP.

5 Conclusions and future work

In this paper we have discussed the integration of a Prolog implementation, tuProlog, into the DCaseLP prototyping environment. Recently, the integration of the DyLOG executable logic-based language into DCaseLP has been designed, thus enriching the set of specification/implementation languages supported by DCaseLP. The integration of tuProlog into DCaseLP represents another step forward in this direction and gives us two main advantages:

1. It allows us to re-use the work previously done with CaseLP regarding the study and the definition of semi-automatic translators from high-level specification languages into Prolog-based communicative agents.
2. It represents a relevant example that we can follow to implement a new DyLOG interpreter in Java, and to integrate this new interpreter into DCaseLP.

Currently, the two advantages above cannot be exploited in practice because we did not have time enough to implement all the components, so we need to make the integration of DyLOG and the languages provided by CaseLP usable. Our future efforts will be channelled in this implementative direction, in order to make DCaseLP the integrated environment for engineering and prototyping heterogeneous MAS that it was intended to be.

References

1. E. Astesiano, M. Martelli, V. Mascardi, and G. Reggio. From Requirement Specification to Prototype Execution: a Combination of a Multiview Use-Case Driven Method and Agent-Oriented Techniques. In J. Debenham and K. Zhang, editors, *Proc. of SEKE'03*. The Knowledge System Institute, 2003.
2. E. Astesiano and G. Reggio. Knowledge Structuring and Representation in Requirement Specification. In *Proc. of SEKE'02*. ACM Press, 2002.
3. E. Astesiano and G. Reggio. Tight Structuring for Precise UML-based Requirement Specifications: Complete Version. Technical Report DISI–TR–03–06, DISI, Università di Genova, Italy, 2003.
4. M. Baldoni, C. Baroglio, L. Giordano, A. Martelli, and V. Patti. Reasoning about communicating agents in the semantic web. In F. Bry, N. Henze, and J. Maluszynski, editors, *Proc. of PPSWR'03*, Springer-Verlag, 2003.
5. M. Baldoni, C. Baroglio, I. Gungui, A. Martelli, M. Martelli, V. Mascardi, V. Patti, and C. Schifanella. Reasoning about agents' interaction protocols inside DCaseLP. In *Proc. of DALT'04*. To appear.
6. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about interaction protocols for web service composition. In M. Bravetti and G. Zavattaro, editors, *Proc. of the WS-FM'04*. Elsevier Science Direct, 2004. Electronic Notes in Theoretical Computer Science.
7. M. Baldoni, C. Baroglio, A. Martelli, and V. Patti. Reasoning about logic-based agent interaction protocols. In this volume.
8. M. Baldoni, L. Giordano, A. Martelli, and V. Patti. Programming rational agents in a modal action logic. Annals of Mathematics and Artificial Intelligence, Special issue on Logic-Based Agent Implementation. To appear.
9. F. Bellifemine, G. Caire, A. Poggi, and G. Rimassa. JADE – a white paper. Available at `http://jade.cselt.it/papers/WhitePaperJADEEXP.pdf`, 2003.
10. FIPA Specifications. `http://www.fipa.org`.
11. Jess home page. `http://herzberg.ca.sandia.gov/jess/`.
12. M. Martelli and V. Mascardi. From UML diagrams to Jess rules: Integrating OO and rule-based languages to specify, implement and execute agents. In F. Buccafurri, editor, *Proc. of AGP'03*, 2003.
13. M. Martelli, V. Mascardi, and F. Zini. CaseLP: a prototyping environment for heterogeneous multi-agent systems. Available at `http://www.disi.unige.it/person/MascardiV/Download/aamas-journal-MMZ04.ps.gz`.
14. J. Odell, H. V. D. Parunak, and B. Bauer. Extending UML for agents. In *Proc. of AOIS'00*, 2000.
15. SICStus Prolog home page. `http://www.sics.se/isl/sicstuswww/site/index.html`.
16. TuProlog home page. `http://lia.deis.unibo.it/research/tuprolog/`.

Communication Architecture in the DALI Logic Programming Agent-Oriented Language*

Stefania Costantini   Arianna Tocchio Alessia Verticchio

Università degli Studi di L'Aquila
Dipartimento di Informatica
Via Vetoio, Loc. Coppito, I-67010 L'Aquila - Italy
{stefcost,tocchio}@di.univaq.it

Abstract. In this paper we describe the communication architecture of the DALI Logic Programming Agent-Oriented language. We have implemented the relevant FIPA compliant primitives, plus others which we believe to be suitable in a logic setting. We have designed a meta-level where: on the one hand the user can specify, via two distinguished primitives tell/told, constraints on communication and/or a communication protocol; on the other hand, meta-rules can be defined for filtering and/or understanding messages via applying ontologies and forms of commonsense and case-based reasoning. These forms of meta-reasoning are automatically applied when needed by a form of reflection.

1 Introduction

Interaction is an important aspect of Multi-agent systems: agents exchange messages, assertions, queries. This, depending on the context and on the application, can be either in order to improve their knowledge, or to reach their goals, or to organize useful cooperation and coordination strategies. In open systems the agents, though possibly based upon different technologies, must speak a common language so as to be able to interact. Agent Communication Languages (ACL), such as the widely-adopted FIPA ACL, provide standardized catalogues of performatives (communication acts), designed in order to ensure interoperability among agent systems [12].

However, beyond standard forms of communication, the agents should be capable of filtering and understanding message contents. A well-understood topic is that of interpreting the content by means of ontologies, which are essentially dictionaries and descriptions that allow different terminologies to be coped with. In a logic language, the use of ontologies can be usefully integrated with forms of commonsense and case-based reasoning, that improve the "understanding" capabilities of an agent. A more subtle point is that it would be useful for an agent to have the possibility to enforce constraints on communication. This implies being able to accept or refuse or rate a message, based on various conditions like for instance the degree of trust in the sender. This also implies to be able to follow a communication protocol in "conversations". Since the degree of trust, the protocol, the ontology, and other factors, can vary with the context,

* We acknowledge support by the *Information Society Technologies programme of the European Commission, Future and Emerging Technologies* under the IST-2001-37004 WASP project.

or can be learned from previous experience, in a logic language agent should and might be able to perform meta-reasoning on communication, so as to interact flexibly with the "external world."

This paper presents the communication architecture of the DALI language. DALI is an Agent-Oriented Logic Programming language designed for executable specification of logical agents, that allows one to define one or more agents interacting among themselves, with other software entities and with an external environment. A main design objective for DALI has been that of introducing in a declarative fashion all the essential features, while keeping the language as close as possible to the syntax and semantics of the traditional Horn-clause language. In practice, most Prolog programs can be understood as DALI programs. Special atoms and rules have been introduced for representing: external events, to which the agent is able to respond (reactivity); actions (reactivity and proactivity); internal events (previous conclusions which can trigger further activity); past and present events (to be aware of what has happened), goals (that the agent can reach). Then, on the line of the arguments proposed in [10], DALI is an enhanced logic language with fully logical semantics [5], that integrates rationality and reactivity, where an agent is able of both backwards and forward reasoning, and has the capability to enforce "maintenance goals" that preserve her internal state, and "achievement goal" that pursue more specific objectives. An extended resolution is provided, so that the DALI interpreter is able to answer queries like in the plain Horn-clause language, but is also able to cope with the different kinds of events.

We have introduced in DALI a communication architecture that specifies in a flexible way the rules of interaction among agents, according to the above-mentioned criteria. The various aspects are modeled in a declarative way, are adaptable to the user and application needs, and can be easily composed. Basically, DALI agents communicate via FIPA ACL, augmented with some primitives which are suitable for a logic language. As a first layer of the architecture, we have introduced a check level that filters the messages. This layer verifies that the message respects the communication protocol of the agent, as well as some domain-independent coherence properties. Several other properties to be checked can be however additionally specified, by expanding the definition of the distinguished predicates *tell/told*. If the message does not pass the check, it is deleted and does not produce any effect. As a second layer, meta-level reasoning is exploited so as to try to understand messages coming from other software entities by using ontologies, and forms of commonsense reasoning.

In summary, when a DALI agent receives a message by another agent, the message is submitted to a check level that controls if it respects the communication protocol and the conditions expressed by the check rules. If the message gets over this control, the agent invokes meta-level reasoning in order to understand its content. The meta-reasoning process uses the agent's ontology and other properties of the terms occurring in the message.

It is important to notice that the definition of the enhanced DALI/FIPA ACL is imported by the agent's code as a library, so as in principle DALI agents may adopt different communication protocols. Also, checks and constraints on communication can be modified without affecting (or without even knowing) the agent's code. The layers of message check and understanding have a predefined default part. However, as

mentioned before they can be extended, improved and adapted to the specific user or application needs by adding rules to the definition of some distinguished predicated.

In this paper we will not be concerned with formal aspects. Rather, we mean to illustrate the communication architecture, and to demonstrate its usefulness mainly by means of significant examples.

The paper is organized as follows. We start by shortly describing the main features of DALI in Section 2. In Section 3 we discuss the new DALI/FIPA protocol and in Section 4 we introduce DALI communication filter. Then, in Sections 5 and 6 we summarize the meta-reasoning layer and how the new architecture works. In Section 7 we show an example of communication between DALI agents, and then conclude in Section 8 with some final remarks.

2 The DALI language

DALI [4] [5] is an Active Logic Programming language designed for executable specification of logical agents. A DALI agent is a logic program that contains a particular kind of rules, reactive rules, aimed at interacting with an external environment. The environment is perceived in the form of external events, that can be exogenous events, observations, or messages by other agents. In response, a DALI agent can perform actions, send messages, invoke goals. The reactive and proactive behavior of the DALI agent is triggered by several kinds of events: external events, internal, present and past events. It is important to notice that all the events and actions are timestamped, so as to record when they occurred. The new syntactic entities, i.e., predicates related to events and proactivity, are indicated with special postfixes (which are coped with by a preprocessor) so as to be immediately recognized while looking at a program.

2.1 External Events

The external events are syntactically indicated by the postfix *E*. When an event comes into the agent from its "external world", the agent can perceive it and decide to react. The reaction is defined by a reactive rule which has in its head that external event. The special token $:>$, used instead of $:-$, indicates that reactive rules performs forward reasoning. E. g., the body of the reactive rule below specifies the reaction to the external event *bell_ringsE* that is in the head. In this case the agent performs an action, postfix *A*, that consists in opening the door.

$bell_ringsE :> open_the_doorA.$

The agent remembers to have reacted by converting the external event into a *past event* (time-stamped).

Operationally, if an incoming external event is recognized, i.e., corresponds to the head of a reactive rule, it is added into a list called EV and consumed according to the arrival order, unless priorities are specified. Priorities are listed in a separate file of directives, where (as we will see) the user can "tune" the agent's behaviour under several respect. The advantage introducing a separate initialization file is that for modifying the directives there is no need to modify (or even to understand) the code.

2.2 Internal Events

The internal events define a kind of "individuality" of a DALI agent, making her proactive independently of the environment, of the user and of the other agents, and allowing her to manipulate and revise her knowledge [3]. An internal event is syntactically indicated by the postfix *I*, and its description is composed of two rules. The first one contains the conditions (knowledge, past events, procedures, etc.) that must be true so that the reaction (in the second rule) may happen.

Internal events are automatically attempted with a default frequency customizable by means of directives in the initialization file. The user's directives can tune several parameters: at which frequency the agent must attempt the internal events; how many times an agent must react to the internal event (forever, once, twice,...) and when (forever, when triggering conditions occur, ...); how long the event must be attempted (until some time, until some terminating conditions, forever).

For instance, consider a situation where an agent prepares a soup that must cook on the fire for K minutes. The predicates with postfix *P* are past events, i.e., events or actions that happened before, and have been recorded. Then, the first rule says that the soup is ready if the agent previously turned on the fire, and K minutes have elapsed since when she put the pan on the stove. The goal *soup_ready* will be attempted from time to time, and will finally succeed when the cooking time will have elapsed. At that point, the agent has to react to this (by second rule) thus removing the pan and switching off the fire, which are two actions (postfix *A*).

$$\begin{aligned} soup_ready :- \ & turn_on_the_fireP, put_pan_on_the_stoveP : T, \\ & cooking_time(K), time_elapsed(T, K). \\ soup_readyI :> \ & take_off_pan_from_stoveA, turn_off_the_fireA. \end{aligned}$$

A suitable directive for this internal event can for instance state that it should be attempted every 60 seconds, starting from when *put_the_pan_on_the_stove* and *turn_on_the_fire* have become past events.

Similarly to external events, internal events which are true by first rule are inserted in a set IV in order to be reacted to (by their second rule). The interpreter, interleaving the different activities, extracts from this set the internal events and triggers the reaction (again according to priorities). A particular kind of internal event is the *goal*, postfix *G*, that stop being attempted as soon as it succeeds for the first time.

2.3 Present Events

When an agent perceives an event from the "external world", it doesn't necessarily react to it immediately: she has the possibility of reasoning about the event, before (or instead of) triggering a reaction. Reasoning also allows a proactive behavior. In this situation, the event is called present event and is indicated by the suffix *N*.

2.4 Actions

Actions are the agent's way of affecting her environment, possibly in reaction to an external or internal event. In DALI, actions (indicated with postfix *A*) may have or not

preconditions: in the former case, the actions are defined by actions rules, in the latter case they are just action atoms. An action rule is just a plain rule, but in order to emphasize that it is related to an action, we have introduced the new token :<, thus adopting the syntax $action :< preconditions$. Similarly to external and internal events, actions are recorded as past actions.

2.5 Past events

Past events represent the agent's "memory", that makes her capable to perform its future activities while having experience of previous events, and of its own previous conclusions. As we have seen in the examples, past event are indicated by the postfix *P*. For instance, *alarm_clock_ ringsP* is an event to which the agent has reacted and which remains in the agent's memory. Each past event has a timestamp T indicating when the recorded event has happened. Memory of course is not unlimited, neither conceptually nor practically: it is possible to set, for each event, for how long it has to be kept in memory, or until which expiring condition. In the implementation, past events are kept for a certain default amount of time, that can be modified by the user through a suitable directive in the initialization file. Implicitly, if a second version of the same past event arrives, with a more recent timestamp, the older event is overridden, unless a directive indicates to keep a number of versions.

3 DALI/FIPA Agent Communication Language

An agent communication language (ACL) is a set of primitives and rules that guide the interaction among several agents [7]. There are a number of standardized languages that the agents can use for communication. The most widely acknowledged is the FIPA ACL, which for the sake of interoperability we have adopted (with few extensions) for DALI. The specification of FIPA messages has the following structure:

- **receiver:** name of the agent that receives the message;
- **language:** the language in which the message is expressed;
- **ontology:** the vocabulary of the words in the message, or, more generally, the description of conceptual relationships between terms and sentences of the same domain, which are expressed in a different terminology;
- **sender:** name of the agent that sends the message;
- **content:** the content of the message, which is the main part, discussed in detail below.

In DALI, a message which has to be sent has the format:

$primitive(content)$

where *primitive* is what is called a *communication performative*, i.e., the specification of the intended meaning of the message, which is then further specified by *content*. For instance, *propose(content)* is a performative aimed at asking another agent to do something, where what should be done is specified by *content*. The DALI interpreter

automatically adds the missing FIPA parameters, thus creating the structure which is actually sent, i.e.:

$$message(\ receiver_address, receiver_name, sender_address, sender_name, \\ language, ontology, primitive(content, sender))$$

Symmetrically, from a message which is received the interpreter extracts the part *primitive(Content,Sender)*, that is what the receiver agent has to consider. In most cases, the receiver will record the item *primitive(Content,Sender)* as a past event. Please notice that *content* may be a conjunction, as the FIPA performatives have different arities.

In the rest of this section, we illustrate the main performatives we adopt. In brackets we indicate if the primitive is FIPA or if it is peculiar of DALI. In most cases, the receiver will record the item *primitive(Content,Sender)* as a past event.

send_message (DALI)

$$send_message(external_event, sender_agent)$$

The act of sending a message to a DALI agent that the receiver will perceive as the communication that the given external event has happened.

propose (FIPA)

$$propose(action, [precondition_1, ..., precondition_n], sender_agent).$$

The act of asking another agent to perform a certain action, given certain preconditions. A DALI agent accepts a proposal if the preconditions are all true, else she rejects the proposal.

accept_proposal (FIPA)

$$accept_proposal(\ action_accepted, [condition_1, ..., condition_n], sender_agent) or \\ accept_proposal(\ action_accepted, [condition_1, ..., condition_n], in_response_to(), \\ sender_agent)$$

The action of accepting a previously received proposal to perform an action where the the conditions of the agreement are enclosed.

reject_proposal (FIPA)

$$reject_proposal(\ action_rejected, [reason_1, ..., reason_n], sender_agent) or \\ reject_proposal(\ action_rejected, [reason_1, ..., reason_n], in_response_to(), \\ sender_agent)$$

The action of rejecting a proposal to perform some action during a negotiation, listing the reasons for rejection.

failure (FIPA)

$$failure(action_failed, motivation, sender_agent)$$

The action of telling another agent that an action was attempted but the attempt failed, enclosing the motivation of the failure.

cancel (FIPA)

$$cancel(action_to_cancel, sender_agent).$$

The action of canceling some previously requested action which had a temporal extent (i.e. cannot be instantaneous).

execute_proc (DALI)

$execute_proc(call_procedure, sender_agent).$

The act of invoking a procedure inside a DALI program.

query_ref (FIPA)

$query_ref(property, N, sender_agent)$

The action of asking another agent for the object referred to by an expression containing free variables. *Property* is the string on which the matching is attempted, and N is the requested number of matches for the object to be identified.

inform (FIPA)

$inform(something, sender_agent)$ *or*
$inform(primitive, values/motivation, sender_agent)$

The sender informs the receiver that a certain "something" is happened.

is_a_fact (DALI)

$is_a_fact(proposition, sender_agent)$

The act of asking if the proposition indicated in the primitive is true.

refuse (FIPA)

$refuse(action_refused, motivation, sender_agent)$

The action of refusing to perform a given action, and of explaining the reason for the refusal. For example,a DALI agent refuses to do an action if the preconditions of the corresponding active rules in the DALI logic program are false. The refusal is recorded as a past event.

confirm (FIPA)

$confirm(proposition, sender_agent)$

The sender informs the receiver that a given proposition is true, where the receiver is supposed to be uncertain about the proposition. The proposition is asserted as a past event.

disconfirm (FIPA)

$disconfirm(proposition, sender_agent)$

The sender informs the receiver that a given proposition is false, where the receiver is instead supposed to believe that the proposition is true. Then, this proposition is deleted from past events.

4 DALI communication filter

In real applications, the interaction between agents raises the problem of security. If an agent is not sufficiently self-defending, she can suffer from damages to either her knowledge base or her rules. It may happen in fact that an agent sends to another one a message with a wrong content, intentionally or not, thus potentially bringing a serious damage. How to recognize a correct message? And a wrong message?

The solution adopted in DALI is aimed at providing a tool for coping, as far as possible, with these problems. When a message is received, it is examined by a check layer composed of a structure which is adaptable to the context and modifiable by the user. This filter checks the content of the message, and verifies if the conditions for the reception are verified. If the conditions are false, this security level eliminates the supposedly wrong message. We have constrained the reception of messages by restricting the range of allowed utterances to the FIPA/DALI primitives, according to additional conditions defined by the user, or, in perspective, learned by the agent herself. For instance, filtering conditions can be based upon reliability of the sender agent. The DALI filter is specified by means of meta-level rules defining the distinguished predicates *tell* and *told*. These meta-rules are contained in a separate file, and can be changed without affecting or even knowing the DALI code. Then, communication in DALI is elaboration-tolerant with respect to both the protocol, and the filter.

4.1 Filter for the incoming messages

Whenever a message is received, with content part *primitive(Content,Sender)* the DALI interpreter automatically looks for a corresponding *told* rule, which is of the form:

$told(Sender, primitive(Content)) :- constraint_1, \ldots, constraint_n.$

where $constraint_i$ can be everything expressible either in Prolog or in DALI. If such a rule is found, the interpreter attempts to prove told(Sender, primitive(Content)). If this goal succeeds, then the message is accepted, and primitive(Content)) is added to the set of the external events incoming into the receiver agent. Otherwise, the message is discarded. Semantically, this can be understood as implicit reflection up to the filter layer, followed by a reflection down to whatever activity the agent was doing, with or without accepting the message. For a detailed and general semantic account of this kind of reflection, the reader may refer to [1].

Below we propose a number of examples of filtering rules. Notice that each agent can have her own set of filtering rules. Since she takes these rules from a separate file, her filtering criteria can vary (by importing a different file) according to the context she is presently involved into.

The following rule accepts a *send_message* primitive if the receiver agent remembers (presumably from past experience) that the sender is reliable, and believes that the content is worth knowing.

$told(\ Sender_agent, send_message(External_event)) :-$
$\quad not(unreliableP(Sender_agent)), interesting(External_event).$

Similarly, the request of either executing a procedure or asserting a fact is taken into consideration if the agent who is asking us is reliable, and in the former case if we actually have the code of that procedure, in the latter case if we are interested in the new fact.

$told(\ Sender_agent, execute_proc(Procedure)) :-$
$\quad not(unreliableP(Sender_agent)), know(Procedure).$
$told(\ Sender_agent, is_a_fact(Proposition)) :-$
$\quad not(unreliableP(Sender_agent)), interesting(Proposition).$

A *query_ref* is acceptable, according to the constraint below, if the Sender agent is reliable and friendly.

$$told(\ Sender_agent, query_ref(Proposition, 3)) :- \\ not(unreliableP(Sender_agent)), friendly(Sender_agent).$$

The agent accepts a *confirm* primitive if the Sender is reliable and the proposition is consistent with her knowledge base. The proposition is recorded as a past event and kept, according to the directive specified in this rule, for 200 seconds. Vice versa, the agent disconfirms a proposition that she knows if the Sender is reliable.

$$told(\ Sender_agent, confirm(Proposition), 200) :- \\ not(unreliableP(Sender_agent)), \\ consistent_with_knowledge_base(Proposition). \\ told(\ Sender_agent, disconfirm(Proposition)) :- \\ not(unreliableP(Sender_agent)), in_knowledge_base(Proposition).$$

The proposal to do an action is acceptable if the agent is specialized for the action and the Sender is reliable.

$$told(\ Sender_agent, propose(Action, Preconditions)) :- \\ not(unreliableP(Sender_agent)), specialized_for(Action).$$

The following constraint checks if the communication protocol is respected. An agent in fact can receive an *accept_proposal* only in response to *propose*. The agent remembers as a past event (for 200 seconds) that she has accepted the proposal to perform an action. This information can be used by an internal event for further inferences.

$$told(\ Sender_agent, accept_proposal(Action, Conditions), \\ in_response_to(Message), 200) :- \\ not(unreliableP(Sender_agent)), \\ functor(Message, F, _), F = propose.$$

We have a similar approach for the other FIPA/DALI primitives *reject_proposal*, *failure*, *refuse* and *inform*.

As the previous examples may have suggested, this model allows one to integrate into the filtering rules the concept the degree of trust. Trust derives from the credibility of the beliefs and of their sources, from the sources' number, convergence, and reliability. All those parameters are easily expressible in the *told* rules. Finally, we emphasize how this communication filter can express constraints not only for a generic communication primitive but also for different contents of the same primitive. For example, we can write:

$$told(Sender_agent, confirm(love_me(julie)), forever) :- \\ not(unreliableP(Sender_agent)), ...$$

When a message is deleted, the DALI interpreter displays the primitive that has not been accepted and the reason on the operator console. The console is a special window which is used to activate/stop agents. The user can optionally keep it open in order either to monitor the agents behaviour, or to participate to the conversation by sending messages to the agents.

4.2 Filter layer for outcoming messages

Symmetrically to *told* rules, the messages that an agent sends are subjected to a check via *tell* rules. There is, however, an important difference: the user can choose which messages must be checked and which not. The choice is made by setting some parameters in the initialization file. The syntax of a *tell* rule is:

$tell(Receiver, Sender, primitive(Content)) : -constraint_1, \ldots, constraint_n$

For every message that is being sent, the interpreter automatically checks whether an applicable *tell* rule exists. If so, the message is actually sent only upon success of the goal $tell(Receiver, Sender, primitive(Content))$.

Below we show as an example two of the default rules coping with the DALI/FIPA primitives. The first *tell* rule authorizes the agent to send the message with the primitive *inform* if the receiver is active in the environment and is presumably interested to the information: via rules like this one we can considerably reduce useless exchange of messages. According to the second rule, the agent sends a *refuse* only if the requested primitive is *is_a_fact* or *query_ref*.

$tell(\ Agent_To, Agent_From, inform(Proposition)) : -$
$\quad active_in_the_world(Agent_To),$
$\quad specialized(Agent_To, Specialization),$
$\quad related_to(Specialization, Proposition).$
$tell(\ Agent_To, Agent_From, refuse(Something, Motivation)) : -$
$\quad arg(1, Something, Primitive),$
$\quad functor(Primitive, F), (F = is_a_fact; F = query_ref).$

5 Meta-reasoning layer

In heterogeneous Multi-agent Systems, in general not all the components speak the same language, and not all of them use the same words to express a concept. The agent that doesn't understand a proposition can either accept the defeat and ignore the message, or try to apply a reasoning process in order to interpret the message contents. The latter solution can be more easily put at work by taking advantage of meta-reasoning capabilities of a logic language. In fact, the use of *ontologies*, which are dictionaries of equivalent terms, can be integrated with several kinds of commonsense reasoning. The ontology of a DALI agent is in a file .txt containing equivalent terms and other properties useful in the meta-reasoning process. E.g., agent *bob*'s ontology can be the following, where *symmetric* is a property of relations, which is asserted to hold of predicate *friend*), and allows him to conclude both *friend(bob,lucy)* and *friend(lucy,bob)* even if originally he could derive only one. The name of the agent enclosed to each item of the ontology allows a group of agents to use the same ontology file, though sharing the contents only partially.

$ontology(bob, rain, water_falling_from_sky).$
$ontology(bob, friend, amico).$
$\ldots\ symmetric(friend).$

Each DALI agent is provided (again in a separate file) with a distinguished procedure called *meta*, to support the meta-reasoning process. This procedure by default includes a number of rules for coping with domain-independent standard situations. The user can add other rules, thus possibly specifying domain-dependent commonsense reasoning strategies for interpreting messages, or implementing a learning strategy to be applied when all else fails. Below we report some of the default meta-reasoning rules that apply the equivalences listed in the ontology, and possibly also exploit symmetry (for binary predicates only):

$meta(\ Initial_term, Final_term, Agent_Sender) :-$
$\quad clause(agent(Agent_Receiver), _),$
$\quad functor(Initial_term, Functor, Arity), Arity = 0,$
$\quad ((ontology(Agent_Sender, Functor, Equivalent_term);$
$\quad ontology(Agent_Sender, Equivalent_term, Functor));$
$\quad (ontology(Agent_Receiver, Functor, Equivalent_term);$
$\quad ontology(Agent_Receiver, Equivalent_term, Functor))),$
$\quad Final_term = Equivalent_term.$
$\ldots$
$meta(\ Initial_term, Final_term, Agent_Sender) :-$
$\quad functor(Initial_term, Functor, Arity), Arity = 2,$
$\quad symmetric(Functor), Initial_term = ..List,$
$\quad delete(List, Functor, Result_list),$
$\quad reverse(Result_list, Reversed_list),$
$\quad append([Functor], Reversed_list, Final_list),$
$\quad Final_term = ..Final_list.$

The procedure meta is automatically invoked, again via reflection, by the interpreter. It is necessary to avoid unwanted variable bindings while meta-reasoning about messages. In DALI, message contents are always reified, i.e., transformed into a ground term, before they are sent, and thus they are received in reified form. Then, the interpreter includes facilities for "reification", or "naming", and "de-reification", or "unnaming" of language expressions (also this issue is discussed at length in [1]).

6 DALI Communication Architecture

In this section we summarize the overall DALI communication architecture, that puts together the functionalities of filter, meta-reasoning and the protocol layers. The architecture consists of three levels: the first level implements the protocol and the filter, i.e., the first two layers of the communication structure; the second level includes the meta-reasoning layer; the third level consists of the DALI interpreter, which is able to activate the agents. Each agent is defined by a .txt file, containing the agent code written in DALI. When an agent receives an external event through the primitive *send_message*, the DALI interpreter calls the filter layer by invoking its internal rule:

$receive(\ send_message(External_event, Agent_Sender)) :-$
$\quad told(Agent_Sender, send_message(External_event)), \ldots$

If the message overcomes the security check, then the interpreter automatically invokes the meta-level in order to understand the external event. The meta reasoning pro-

cess initially has to "un-name" the message content, so as to verify if (an instance of) *External_event* belong to the set of external events known by the agent without applying the meta procedure. If it is the case, then the agent reacts directly, else the interpreter takes advantage of meta reasoning capabilities for finally finding a reactive rule of the logic program applicable to the context.

Similarly, the meta reasoning level is called for the primitives: *propose*, *execute_proc*, *query_ref* and *is_a_fact*. The procedure *meta* in fact contains a special rule for each different communication act. E. g., the code for the primitive *propose* is reported below: if the action proposed belong to the actions occurring in the logic program of agent, then the interpreter sends back to the proposer agent a message *accept_proposal*, else a *reject_proposal*.

$$
\begin{array}{ll}
receive(propose(& Action, Conditions, Sender_Agent)) :- \\
 & told(Sender_Agent, propose(Action, Conditions)), \ldots, \\
 & call_meta_propose(Action, Conditions, Sender_Agent). \\
call_meta_propose(& Action, Conditions, Sender_Agent) :- \\
 & once(call_propose(Action, Conditions, Sender_Agent)). \\
call_propose(& Action, Conditions, Sender_Agent) :- \\
 & denaming(Action, New_action), exists_action(New_action), \\
 & execute_propose(New_action, Conditions, Sender_Agent). \\
call_propose(& Action, Conditions, Sender_Agent) :- \\
 & meta(Action, New_term, _), denaming(New_term, New_action), \\
 & exists_action(New_action), \\
 & execute_propose(New_action, Conditions, Sender_Agent).
\end{array}
$$

In the first rule of *call_propose* the interpreter, after the de-naming of the term, verifies whether the agent knows the action, without applying either the ontology or other properties. In the second rule, called if the first one fails, the interpreter changes the name of the action by applying the meta reasoning and, after the de-naming, checks again if (an instance of) the required action exists among those feasible by the agent. The link with the part of the DALI interpreter that handles events, goals and actions is represented by the procedure *execute_propose*. Via this procedure the action, if recognized, is put into the queue of actions that are waiting to be executed.

7 Example: an Italian client in an English pub

We propose an example of interaction between DALI agents employing the communication architecture that we have discussed. We consider four agents: (i) **agent waiter:** he is a pub (or cafeteria) waiter that receives orders and serves drinks to clients; (ii) **agent *gino*:** an italian client, who walks into the cafeteria and orders a beer incorrectly; (iii) **agent *wife*:** wife of another client, **bob**, who is a drunkard and never finds his way home; (iv) **agent *friend*:** friend of the italian client *gino*.

The italian client *gino* walks into the pub and orders a beer, mixing Italian and English languages and mispelling the word *beer*. He sends to the waiter the message:

$$send_message(voglio(gino, ber), gino).$$

The waiter speaks a bit of italian, i.e., he applies the item of his ontology: $ontology(_, voglio, request)$, and understands that *voglio* is equivalent to *request*. But he still doesn't understand *ber*, and thus informs *gino* about this problem.

$$
\begin{aligned}
¬_know(C,P) :- \; requestP(C,P), not(clause(product(P,_),_)).\\
¬_know(C,P) :- \; requestP(C,_,P), not(clause(product(P,_),_)).\\
¬_knowI(C,P) :> clause(agent(A),_),\\
&\qquad messageA(C, inform(not_know(P),A)).
\end{aligned}
$$

gino, not speaking English very well, asks friend for how to formulate his request correctly:

$$
\begin{aligned}
&ask_to_friend(F) :- \; informP(not_know(F), waiter).\\
&ask_to_friendI(F) :> clause(agent(Agent),_),\\
&\qquad messageA(friend, send_message(how_tell(F,Agent),Agent)).
\end{aligned}
$$

The agent *friend*, aware of the poor English of *gino*, by using the ontology he has learned during their acquaintance (that thus contains the fact *ontology(gino,ber,beer)*), informs the waiter that *ber* is equivalent to *beer*.

$$
\begin{aligned}
&how_tellE(F,Agent) :> clause(agent(Ag),_),\\
&\qquad clause(ontology(Ag,F,P1),_),\\
&\qquad messageA(waiter, inform(how_tell(Agent,F,P1),Ag)).
\end{aligned}
$$

The waiter, when receives the information about the term *ber* adds it to his ontology and serves the beer (if available).

$$
\begin{aligned}
&request(Agent,F,P1) :- \; informP(how_tell(Agent,F,P1),_).\\
&requestI(Agent,F,P1) :- assert(ontology(Agent,F,P1)).\\
&serve_drink(C,F) :- \; requestP(C,_,F), available(F).\\
&serve_drinkI(C,F) :> \; serveA(C,F).
\end{aligned}
$$

The agent *wife*, if the husband *bob* isn't back home by 11 p.m., tries to go to the pub in order to find out if *bob* is there. She asks the waiter (by using the primitive is_a_fact) if *bob* is in the pub. The waiter responds by the primitive *inform*, according to the DALI/FIPA protocol.

$$
\begin{aligned}
¬_at_home(Husband,Today) :- \; missing_husbandP(Husband),\\
&\qquad datime(T), arg(3,T,Today),\\
&\qquad arg(4,T,Hour), Hour >= 23.\\
¬_at_homeI(Husband,Today) :> clause(agent(Agent),_), go_to_pubA,\\
&\qquad messageA(waiter,\\
&\qquad is_a_fact(in_pub(M,Today),Agent)).
\end{aligned}
$$

Then, by exploiting the content of the primitive *inform* sent back by the waiter (who records all clients that enter and exit the pub), knows that the husband is at the pub. She reacts by screaming, taking her husband home and telling to the waiter that he mustn't serve alcoholic drinks to her husband (including wine).

$$
\begin{aligned}
&husband_in_pub :- informP(agree(in_pub(gino,9)), values(yes), waiter),\\
&\qquad messageA(waiter, inform(not_serve(gino,wine),wife)),\\
&\qquad messageA(waiter, confirm(alcoholic(wine),wife)).
\end{aligned}
$$

The *inform* and *confirm* primitives sent to the waiter are used by the told rule of waiter's check layer:

$told(\ Ag, send_message(request(Ag, P))) : -$
$\quad not(informP(not_serve(Ag, P), wife)), alcoholicP(P).$

When the drunkard husband will ask for an alcoholic drink in the future, the message will be eliminated.

8 Conclusions

This paper has discussed how communication has been designed and implemented in the DALI Logic Programming Agent-Oriented language. We have shown the kinds of interaction and the abstract roles that the DALI/FIPA protocol supports and the functionalities of the check and meta-reasoning layers. We have noticed that the proposed architecture takes profit of features which are proper of logic languages, such as meta-reasoning and logical reflection.

There are other FIPA-compliant agent frameworks. A future aim of our experiments in fact is that of ensuring interoperability between DALI and these other approaches. A relevant one is JADE, a FIPA-compliant framework fully developed in Java. Each agent platform, written in Java and importing the JADE libraries, can be split on several hosts. Each agent is implemented as a Java thread, and Java events are used for effective and light-weight communication between agents on the same host. A number of FIPA-compliant DFs (Directory Facilitators) agents can be started at run time in order to implement multi-domain applications. About security, JADE provides proper mechanisms to authenticate and verify the rights assigned to agents. [2]

The 3APL platform is the first platform that has supported easy and direct implementation and execution of cognitive agents. The platform can be distributed across different machines connected in a network. Moreover, the 3APL platform is FIPA compliant in the sense that agents running on this platform can in principle communicate with agents that run on a different FIPA compliant platforms such as JADE. [11]

However, the DALI project demonstrates that a logic programming language with a logic semantics [5] is able to exhibit, also for communication, features that are as powerful (and even more flexible) as those of approaches that are either not fully logical, or semantically more complex. The DALI implementation cannot currently compete in efficiency with others like JADE, which have been developed in the industry, and on which a lot of effort has been spent by several companies and universities. Nevertheless, DALI is competitive from the point of view of the ease and flexibility of use, for every kind of application, but especially where context-sensitivity, adaptability and intelligence are needed. We have equipped DALI with an interface with Java, that has allowed us to develop applications (namely in component management and reconfiguration in distributed systems [3]) where the Java part is able to interact at a low level with legacy systems, and the DALI part implements intelligent reasoning and sophisticated interaction. The declarative semantics of DALI has been defined in [5]. The operational semantics is being defined in a Ph.D. Thesis. The behaviour of DALI interpreter has been modeled and checked by using the Murϕ model checker [9]. A future aim of this research is that of developing and experimenting cooperative models for DALI logical agents, also based on game theory. As a first step, we are studying formal models for

making DALI agents adaptive with respect to the level of trust that they assign to the other agents.

References

1. J. Barklund, S. Costantini, P. Dell'Acqua e G. A. Lanzarone, *Reflection Principles in Computational Logic*, Journal of Logic and Computation, Vol. 10, N. 6, December 2000, Oxford University Press, UK.
2. F. Bellifemine, A. Poggi, G. Rimassa, *JADE: A FIPA-compliant agent framework*, In: *Proc. of the 4th International Conference and Exhibition on The Pratical Application of Intelligent Agents and Multiagents*, held in London,UK, December 1999.
3. M. Castaldi, S. Costantini, S. Gentile, A. Tocchio, *A Logic-Based Infrastructure for Reconfiguring Applications*, In: J. A. Leite, A. Omicini. L. Sterling, P. Torroni (eds.), *Declarative Agent Languages and Technologies, Proc. of the 1st International Workshop, DALT 2003* (held in Melbourne, Victoria, July 2003), Available also on-line, URL http://centria.di.fct.unl.pt/ jleite/dalt03/papers/dalt2003proceedings.pdf.
4. S. Costantini. Towards active logic programming. In A. Brogi and P. Hill, editors, *Proc. of 2nd International Workshop on component-based Software Development in Computational Logic (COCL'99)*, PLI'99, (held in Paris, France, September 1999), Available on-line, URL http://www.di.unipi.it/ brogi/ResearchActivity/COCL99/proceedings/index.html.
5. S. Costantini and A. Tocchio, *A Logic Programming Language for Multi-agent Systems*, In S. Flesca, S. Greco, N. Leone, G. Ianni (eds.), *Logics in Artificial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002*, (held in Cosenza, Italy, September 2002), LNAI 2424, Springer-Verlag, Berlin, 2002.
6. U. Endriss, N. Maudet, F. Sadri, F. Toni, *Logic-based Agent Communication Protocols*, In: *Lecture Notes in Computer Science, Advances in Agent Communication, Proc. of the International Workshop on Agent Communication Languages ACL03*, (held in Melbourne, Australia, July 14 2003), LNCS 2922, Springer-Verlag, Berlin, 2004.
7. M.-P. Huget, J.-L. Koning, *Interaction Protocol Engineering*, In: *Lecture Notes in Computer Science, Communication in Multiagent Systems, Agent Communication Languages and Conversation Policies*, LNCS 2650, Springer-Verlag, Berlin, 2003.
8. M. N. Huhns, L. M. Stephens, *Multiagent System and Societies of Agents*, In: *Multiagent Systems: A modern Approach to Distributed Artificial Intelligence*, 1999.
9. B. Intrigila, I. Melatti, A. Tocchio, *Model-checking DALI with Murϕ*, Tech. Rep., Univ. of L'Aquila, 2004.
10. R. A. Kowalski, *How to be Artificially Intelligent - the Logical Way*, Draft, revised February 2004, Available on line, URL http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/rak.html.
11. E.C. Van der Hoeve, M. Dastani, F. Dignum, J.-J. Meyer, *3APL Platform*, In: *Proc. of the The 15th Belgian-Dutch Conference on Artificial Intelligence(BNAIC2003)*, held in Nijmegen, The Netherlands, 2003.
12. J. M. Serrano, S. Ossowski. *An Organisational Approach to the Design of Interaction Protocols*, In: *Lecture Notes in Computer Science, Communications in Multiagent Systems: Agent Communication Languages and Conversation Policies*, LNCS 2650, Springer-Verlag, Berlin, 2003.

Preserving (Security) Properties under Action Refinement*

Annalisa Bossi, Carla Piazza, and Sabina Rossi

Dipartimento di Informatica, Università Ca' Foscari di Venezia
via Torino 155, 30172 Venezia, Italy
{bossi,piazza,srossi}@dsi.unive.it

Abstract. In the design process of distributed systems we may have to replace abstract specifications of components by more concrete specifications, thus providing more detailed design information. In the context of process algebra this well-known approach is often referred to as *action refinement*. In this paper we study the relationships between action refinement, compositionality, and (security) process properties within the Security Process Algebra (SPA). We formalize the concept of action refinement both as a structural inductive definition and in terms of subsequent context compositions. We study compositional properties of our notion of refinement and provide conditions under which general process properties are preserved through it. Finally, we consider information flow security properties and define decidable classes of secure terms which are closed under action refinement.

1 Introduction

In the development of complex systems it is common practice to first describe it succinctly as a simple abstract specification and then refine it stepwise to a more concrete implementation. This hierarchical specification approach has been successfully developed for sequential systems where abstract-level instructions are expanded until a concrete implementation is reached (see, e.g., [21]).

In the context of process algebra, this refinement methodology amounts to defining a mechanism for replacing abstract actions with more concrete processes. We adopt the terminology *action refinement* to refer to this stepwise development of systems specified as terms of a process algebra. We refer to [14] for a survey on the state of the art of action refinement in process algebra.

Action refinement in process algebras is usually defined by extending the syntax with some compositional operator [1, 13]. Here we follow a different approach and instead of extending the language, we use a construction based on context composition. This allows us to reason on the relationships between action refinement and the security properties of SPA processes that we have deeply studied in, e.g., [3]. In the last part of the paper we prove that our definition of action refinement is indeed equivalent to the one presented in [1].

* This work has been partially supported by the EU Contract IST-2001-32617 "Models and Types for Security in Mobile Distributed Systems" (MyThS).

In this paper we model action refinement as a ternary function *Ref* taking as arguments an action r to be refined, a system description E on a given level of abstraction and an interpretation of the action r on this level by a more concrete process F on a lower abstraction level. The refined process can be obtained either by applying a structural inductive definition or through a more complex context composition as described by the following simple example.

Let E be the process $a.r.b.\mathbf{0} + c.\mathbf{0}$ and r be the action we intend to refine by the process $F \equiv d_1.d_2.\mathbf{0}$. The refined process, denoted by $Ref(r, E, F)$, will be the process $a.d_1.d_2.b.\mathbf{0} + c.\mathbf{0}$ which can be obtained in two equivalent ways: (1) we can either apply a structural inductive definition as follows: $Ref(r, E, F) = a.Ref(r, r.b.\mathbf{0}, F) + c.\mathbf{0} = a.F'[b.\mathbf{0}] + c.\mathbf{0}$ where $F'[Y]$ is the context $d_1.d_2.Y$ and $F'[b.\mathbf{0}]$ is the process $d_1.d_2.b.\mathbf{0}$; (2) or we can compute the refinement by a single context composition as $E'[F'[b.\mathbf{0}]]$ where $E'[X]$ is the context $a.X + c.\mathbf{0}$ while $F'[Y]$ is as above the context $d_1.d_2.Y$.

Our definitions follow the static syntactic approach to action refinement (see, e.g., [19]). We prove several compositional properties of our notion of refinement. Indeed, compositional properties are fundamental in the stepwise development of complex systems. They allow us to refine sub-components of the system, while guaranteeing that the final result does not depend on the order in which the refinements are applied. We also provide conditions under which our notion of refinement preserves general properties of processes and, in particular, we focus on security properties.

In system development, it is important to consider security related issues from the very beginning. Indeed, considering security only at the final step could lead to a poor protection or, even worst, could make it necessary to restart the development from scratch. A security-aware stepwise development requires that the security properties of interest are either preserved or gained during the development steps, until a concrete (i.e., implementable) specification is obtained.

In this paper we consider *information flow security* properties (see, e.g., [12, 9, 15]), i.e., properties that allow one to express constraints on how information should flow among different groups of entities. These properties are usually formalized by considering two groups of entities labelled with two security levels: *high* (H) and *low* (L). The only constraint is that no information should flow from H to L. For example, to guarantee confidentiality in a system, it is sufficient to label every confidential (i.e., secret) information with H and then partition each system user as H and L, depending on whether such a user is or is not authorized to access confidential information. The constraint of no information flow from H to L guarantees that no access to confidential information is possible by L-labelled users. We consider the bisimulation-based security property named *Persistent Bisimulation-based non Deducibility on Compositions* (P_BNDC, for short) [10]. Property P_BNDC is based on the idea of Non-Interference [12] and requires that every state which is reachable by the system still satisfies a basic Non-Interference property. We show how to both instantiate and extend the results obtained for general process properties in order to provide decidable conditions ensuring that P_BNDC is preserved under action refinement.

The paper is organized as follows. In Section 2 we recall some basic notions of the SPA language. In Section 3 we formalize our notion of action refinement as a structural inductive definition. We also study its compositional properties. In Section 4 we reformulate action refinement in terms of context composition and we state conditions under which general process properties are preserved through action refinement. In Section 5 we consider the security property P_BNDC and define decidable classes of P_BNDC processes which are closed under action refinement. Finally, in Section 6 we discuss some related works.

2 Basic Notions

The *Security Process Algebra* (SPA) [9] is a variation of Milner's CCS [18] where the set of visible actions is partitioned into two security levels, high and low, in order to specify multilevel systems. SPA syntax is based on the same elements as CCS, i.e.: a set $\mathcal{L} = I \cup O$ of *visible* actions where $I = \{a, b, \ldots\}$ is a set of *input* actions and $O = \{\bar{a}, \bar{b}, \ldots\}$ is a set of *output* actions; a special action τ which models internal computations, not visible outside the system; a function $\bar{\cdot} : \mathcal{L} \rightarrow \mathcal{L}$, such that $\bar{\bar{a}} = a$, for all $a \in \mathcal{L}$. $Act = \mathcal{L} \cup \{\tau\}$ is the set of all *actions*. The set of visible actions is partitioned into two sets, H and L, of high security actions and low security actions such that $\overline{H} = H$ and $\overline{L} = L$, where $\overline{H}$ and $\overline{L}$ are obtained by applying function $\bar{\cdot}$ to all the elements in H and L, respectively.

The syntax of SPA *terms* is as follows[1]:

$$T ::= \mathbf{0} \mid Z \mid a.T \mid T + T \mid T|T \mid T \setminus v \mid T[f] \mid recZ.T$$

where Z is a variable, $a \in Act$, $v \subseteq \mathcal{L}$, $f : Act \rightarrow Act$ is such that $f(\bar{l}) = \overline{f(l)}$ for $l \in \mathcal{L}$, $f(\tau) = \tau$, $f(H) \subseteq H \cup \{\tau\}$, and $f(L) \subseteq L \cup \{\tau\}$. We apply the standard notions of *free* and *bound* (occurrences of) variables in a SPA term. More precisely, all the occurrences of the variable Z in $recZ.T$ are *bound*; while Z is *free* in a term T if there is an occurrence of Z in T which is not bound.

A SPA *process* is a SPA term without free variables. We denote by $\mathcal{E}$ the set of all SPA processes, ranged over by $E, F, \ldots$.

The operational semantics of SPA processes is given in terms of *Labelled Transition Systems* (LTS, for short). In particular, the LTS $(\mathcal{E}, Act, \rightarrow)$, whose states are processes, is defined by structural induction as the least relation generated by the axioms and inference rules reported in Figure 1.

Intuitively, $\mathbf{0}$ is the empty process that does nothing; $a.E$ is a process that can perform an action a and then behaves as E; $E_1 + E_2$ represents the nondeterministic choice between the two processes E_1 and E_2; $E_1|E_2$ is the parallel composition of E_1 and E_2, where executions are interleaved, possibly synchronized on complementary input/output actions, producing the silent action τ; $E \setminus v$ is a process E prevented from performing actions in v; $E[f]$ is the process E whose actions are renamed *via* the relabelling function f; if in T there is

[1] Actually in [9] recursion is introduced through constant definitions instead of the *rec* operator.

Prefix	$\dfrac{-}{a.E \xrightarrow{a} E}$
Sum	$\dfrac{E_1 \xrightarrow{a} E_1'}{E_1 + E_2 \xrightarrow{a} E_1'}$ $\quad \dfrac{E_2 \xrightarrow{a} E_2'}{E_1 + E_2 \xrightarrow{a} E_2'}$
Parallel	$\dfrac{E_1 \xrightarrow{a} E_1'}{E_1 \vert E_2 \xrightarrow{a} E_1' \vert E_2}$ $\quad \dfrac{E_2 \xrightarrow{a} E_2'}{E_1 \vert E_2 \xrightarrow{a} E_1 \vert E_2'}$ $\quad \dfrac{E_1 \xrightarrow{l} E_1' \quad E_2 \xrightarrow{\bar{l}} E_2'}{E_1 \vert E_2 \xrightarrow{\tau} E_1' \vert E_2'}$
Restriction	$\dfrac{E \xrightarrow{a} E'}{E \setminus v \xrightarrow{a} E' \setminus v}$ if $a \notin v$
Relabelling	$\dfrac{E \xrightarrow{a} E'}{E[f] \xrightarrow{f(a)} E'[f]}$
Recursion	$\dfrac{T[recZ.T[Z]] \xrightarrow{a} E'}{recZ.T[Z] \xrightarrow{a} E'}$

with $a \in Act$ and $l \in \mathcal{L}$.

Fig. 1. The operational rules for SPA

at most the free variable Z, then $recZ.T[Z]$ is the recursive process which can perform all the actions of the process obtained by substituting $recZ.T[Z]$ to the place-holder Z in the term $T[Z]$.

The concept of *observation equivalence* is used to establish equalities among processes and it is based on the idea that two systems have the same semantics if and only if they cannot be distinguished by an external observer. This is obtained by defining an equivalence relation over $\mathcal{E}$ equating two processes when they are indistinguishable. In this paper we consider the relations named *weak bisimulation*, $\approx$, and *strong bisimulation*, $\sim$, defined by Milner for CCS [18]. They equate two processes if they are able to mutually simulate their behavior step by step.

We use the following notations: $E \xrightarrow{a} E'$ to denote the transition labelled by a from E to E', $E \stackrel{a}{\Longrightarrow} E'$ to denote any sequence of transitions $E(\xrightarrow{\tau})^* \xrightarrow{a} (\xrightarrow{\tau})^* E'$ where $(\xrightarrow{\tau})^*$ denotes a (possibly empty) sequence of τ labelled transitions, and $E \stackrel{\hat{a}}{\Longrightarrow} E'$ which stands for $E \stackrel{a}{\Longrightarrow} E'$ if $a \in \mathcal{L}$, and for $E(\xrightarrow{\tau})^* E'$ if $a = \tau$. We say that E' is reachable from E if there exist $a_1, \ldots, a_n \in Act$ such that $E \xrightarrow{a_1} \ldots \xrightarrow{a_n} E'$.

Weak bisimulation does not care about internal τ actions while strong bisimulation does.

Definition 1 (Weak and Strong Bisimulation). *A symmetric binary relation* $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ *over processes is a* weak bisimulation *if* $(E, F) \in \mathcal{R}$ *implies, for all* $a \in Act$, *if* $E \xrightarrow{a} E'$, *then there exists* F' *such that* $F \stackrel{\hat{a}}{\Longrightarrow} F'$ *and* $(E', F') \in \mathcal{R}$. *Two processes* $E, F \in \mathcal{E}$ *are* weakly bisimilar, *denoted by* $E \approx F$, *if there exists a weak bisimulation* $\mathcal{R}$ *containing the pair* (E, F).

The definition of strong bisimulation *is obtained by replacing* $\stackrel{\hat{a}}{\Longrightarrow}$ *with* $\xrightarrow{a}$ *in the sentence above. Two processes* $E, F \in \mathcal{E}$ *are* strongly bisimilar, *denoted by* $E \sim F$, *if there exists a strong bisimulation* $\mathcal{R}$ *containing the pair* (E, F).

The relation $\approx$ ($\sim$) is the largest weak (strong) bisimulation and it is an equivalence relation.

A SPA term with free variables can be seen as an environment with holes (the free occurrences of its variables) in which other SPA terms can be inserted. The result of this substitution is still a SPA term, which could be a process. For instance, in the term $h.\mathbf{0}|(l.X + \tau.\mathbf{0})$ we can replace the variable X with the process $\bar{h}.\mathbf{0}$ obtaining the process $h.\mathbf{0}|(l.\bar{h}.\mathbf{0} + \tau.\mathbf{0})$; or we can replace X by the term $a.Y$ obtaining the term $h.\mathbf{0}|(l.a.Y + \tau.\mathbf{0})$. When we consider a SPA term as an environment we call it *context*[2], i.e., a SPA context is a SPA term in which free variables may occur.

Given a context C, we use the notation $C[Y_1, \ldots, Y_n]$ to emphasize the free variables $Y_1, \ldots, Y_n$ occurring in C. The term $C[T_1, \ldots, T_n]$ is obtained from $C[Y_1, \ldots, Y_n]$ by simultaneously replacing all the free occurrences of $Y_1, \ldots, Y_n$ with the terms $T_1, \ldots, T_n$, respectively. For instance, given the contexts $C[X] \equiv h.\mathbf{0}|(l.X + \tau.\mathbf{0})$ and $D[X, Y] \equiv (l.X + \tau.\mathbf{0})|Y$, the notation $C[\bar{h}.\mathbf{0}]$ stands for $h.\mathbf{0}|(l.\bar{h}.\mathbf{0} + \tau.\mathbf{0})$, while the notation $D[\bar{h}.\mathbf{0}, \bar{l}.\mathbf{0}]$ stands for $(l.\bar{h}.\mathbf{0} + \tau.\mathbf{0})|\bar{l}.\mathbf{0}$.

Following [18] we extend binary relations on processes to contexts as follows.

Definition 2 (Relations on Contexts). *Let* $\mathcal{R} \subseteq \mathcal{E} \times \mathcal{E}$ *be a relation over processes. Let* C, D *be two contexts with free variables* $Y_1, \ldots, Y_n$. *We say that* $C[Y_1, \ldots, Y_n]\ \mathcal{R}\ D[Y_1, \ldots, Y_n]$ *if* $C[E_1, \ldots, E_n]\ \mathcal{R}\ D[E_1, \ldots, E_n]$ *for any choice of* $E_1, \ldots, E_n \in \mathcal{E}$. *We also use* $C\ \mathcal{R}\ D$ *to denote* $C[Y_1, \ldots, Y_n]\ \mathcal{R}\ D[Y_1, \ldots, Y_n]$.

As an example, the contexts $C[X, Y] \equiv a.X + \tau.Y$ and $D[X, Y] \equiv a.\tau.X + \tau.Y$ are weakly bisimilar since for all $E, F \in \mathcal{E}$ it holds $a.E + \tau.F \approx a.\tau.E + \tau.F$.

Strong bisimulation is a congruence, i.e., if $C[X] \sim D[X]$ and $E \sim F$, then $C[E] \sim D[F]$. Weak bisimulation is not a congruence, i.e., if two contexts $C[X]$ and $D[X]$ are weakly bisimilar, and two processes E and F are weakly bisimilar, then $C[E]$ and $D[F]$ are not necessarily weakly bisimilar. However, weak bisimulation is a congruence over the *guarded* SPA language whose terms are defined by replacing the production $T + T$ with $a.T + a.T$ in the SPA syntax.

3 Action Refinement

It is standard practice in software development to obtain the final program starting from an abstract, possibly not executable, specification by successive refinement steps. Abstract operations are replaced by more detailed programs which

[2] Notice that a SPA term denotes either a process or a context.

can be further refined, until a level is reached where no more abstractions occur. In the context of process algebra, this stepwise development amounts to interpreting actions on a higher level of abstraction by more complicated processes on a lower level. This is obtained by introducing a mechanism to replace actions by processes. There are several ways to do this. We adopt the syntactic approach and define the refinement step as a syntactic process transformation.

We need to introduce some notation. Given a process F and a variable Y, we denote by $F^0[Y]$ the context obtained by replacing each occurrence of $\mathbf{0}$ in F with the variable Y. As an example, consider the process $F \equiv recZ.(a.Z + b.\mathbf{0})$. Then $F^0[Y] \equiv recZ.(a.Z + b.Y)$.

To introduce our notion of action refinement we also need to define which are the *refinable* actions of a process. This concept is based on the following notions of *bound* and *free* actions.

Definition 3. (Bound and Free actions) *Let T be a SPA term. The set of* bound *actions of T, denoted by $bound(T)$, is inductively defined as follows:*

$$
\begin{array}{l}
bound(\mathbf{0}) = \emptyset \\
bound(Z) = \emptyset \quad \textit{where } Z \textit{ is a variable} \\
bound(a.T) = bound(T) \\
bound(T_1 + T_2) = bound(T_1) \cup bound(T_2) \\
bound(T_1 | T_2) = bound(T_1) \cup bound(T_2) \\
bound(T \setminus v) = bound(T) \cup v \\
bound(T[f]) = bound(T) \cup \{a, f(a) \mid f(a) \neq a\} \\
bound(recZ.T) = bound(T)
\end{array}
$$

An action occurring in T is said to be free *if it is not bound. We denote by $free(T)$ the set of free actions of T.*

In practice, an action is bound in a term T if either it is restricted in a subterm of T or it belongs to the domain or the codomain of a relabelling function f occurring in T. For instance, the actions a and $\bar{a}$ occur bound in the process $E \equiv a.\mathbf{0} + recZ.((\bar{a}.Z + b.a.\mathbf{0}) \setminus \{a, \bar{a}\})$.

An abstract action r occurring in a process E is refinable if r is not bound in E and, in order to avoid problems with synchronizations, $\bar{r}$ does not occur in E. We also require that the process F which is intended to refine r is different from $\mathbf{0}$ and that it does not contain the parallel operator. Moreover, r and $\bar{r}$ should not occur in F otherwise we would enter into an infinite loop of refinements. Finally we impose that the free actions of F are not bound in E and vice-versa, to avoid undesired bindings of actions in the refined process. All these requirements are formalized in the following notion of *refinability*. We will discuss them in the next subsection.

Definition 4. (Refinability) *Let E, F be SPA processes and $r \in \mathcal{L}$. The action r is said to be* refinable in E with F *if:*

(*a*) *F is not the process $\mathbf{0}$;*
(*b*) *F does not contain any occurrence of the parallel operator;*

(c) *$r \notin bound(E)$ and $\bar{r}$ does not occur in E;*
(d) *r and $\bar{r}$ do not occur in F;*
(e) *for all subterm E' of E, $(bound(E') \cap free(F)) \cup (bound(F) \cap free(E')) = \emptyset$*

Example 1. Consider the processes $E \equiv (r.a.\mathbf{0}|\bar{a}.b.\mathbf{0}) \setminus \{a, \bar{a}\}$ and $F \equiv c.d.\mathbf{0}$. In this case action r is refinable in E with F.

Consider now the processes E as above and $F_1 \equiv b.\mathbf{0}+(c.d.\mathbf{0})\setminus\{b\}$. In this case condition (e) of Definition 4 is not satisfied since $bound(F_1) \cap free(E) = \{b\} \neq \emptyset$. Hence r is not refinable in E with F_1. □

The refinement of an abstract action r in a process E with a refining process F is obtained by replacing each occurrence of r in E with F. In order to support action refinement, in the literature the prefixing operator is usually replaced by sequential composition ";" (see [1, 13]). Here we follow a different approach and instead of extending the language, we use a construction based on context composition. Thus, for instance the refinement of the action r in the process $E \equiv a.r.b.\mathbf{0}$ with the process $F \equiv c.d.\mathbf{0}$ is obtained by substituting $b.\mathbf{0}$ for Y in $a.F^0[Y] \equiv a.c.d.Y$, i.e., it is the process $a.F^0[b.\mathbf{0}]$. The conditions on the refinable actions and the fact that F does not contain the parallel operator, ensure that our notion of action refinement is comparable with more classical ones like, e.g., [1] (see Section 6). Moreover, the fact that we do not modify our language, allows us to directly apply our security notions for SPA processes also when reasoning on action refinement.

Our notion of *action refinement* is defined by structural induction on the process to be refined.

Definition 5. (Action Refinement) *Let E, F be SPA processes such that r is an action refinable in E with F. The* refinement of r in E with F *is the process $Ref(r, E, F)$ inductively defined on the structure of E as follows:*

(1) $Ref(r, \mathbf{0}, F) \equiv \mathbf{0}$
(2) $Ref(r, Z, F) \equiv Z$
(3) $Ref(r, r.E_1, F) \equiv F^0[Ref(r, E_1, F)]$
(4) $Ref(r, a.E_1, F) \equiv a.Ref(r, E_1, F), \; if\ a \neq r$
(5) $Ref(r, E_1[f], F) \equiv Ref(r, E_1, F)[f]$
(6) $Ref(r, E_1 \setminus v, F) \equiv Ref(r, E_1, F) \setminus v$
(7) $Ref(r, E_1 + E_2, F) \equiv Ref(r, E_1, F) + Ref(r, E_2, F)$
(8) $Ref(r, E_1|E_2, F) \equiv Ref(r, E_1, F)|Ref(r, E_2, F)$
(9) $Ref(r, recZ.E_1, F) \equiv recZ.Ref(r, E_1, F)$

Point (3) of definition above deals with the basic case in which we replace an occurrence of r with the refining process F. If $E \equiv r.E_1$ and r is the only occurrence of r in E, then $Ref(r, E, F) \equiv F^0[Ref(r, E_1, F)] \equiv F^0[E_1]$ representing the process which first behaves as F and then, when the execution of F is terminated, proceeds as E_1. In all the other cases the refinement process enters inside the components of E. This is correct also when restriction or relabelling operators are involved since conditions (c), (d) and (e) of Definition 4 guarantee

that we never refine restricted or relabelled actions and that undesired bindings of actions will never occur.

Example 2. Let $E \equiv r.a.\mathbf{0} + b.\mathbf{0}$ and $F \equiv c.\mathbf{0} + d.\mathbf{0}$. It is immediate to observe that r is refinable in E with F. By applying points 7. and 3. of Definition 5 we get $\mathit{Ref}(r, E, F) \equiv c.a.\mathbf{0} + d.a.\mathbf{0} + b.\mathbf{0}$.

Let $E \equiv (a.r.b.\mathbf{0}) \setminus \{b\}$ and $F \equiv c.d.\mathbf{0}$. Since $\mathit{bound}(E) = \{b\}$ and b does not occur in F, r is refinable in E with F. By applying points 6., 4. and 3. of our definition of action refinement we get $\mathit{Ref}(r, E, F) \equiv (a.c.d.b.\mathbf{0}) \setminus \{b\}$. Notice that, our notion of refinability does not allow us to refine r in E with $F_1 \equiv b.d.\mathbf{0}$. However, as done in [1], we can first apply an α conversion mapping E into the equivalent process $E_1 \equiv (a.r.e.\mathbf{0}) \setminus \{e\}$ and then refine r in E_1 with F_1 getting the expected process $(a.b.d.e.\mathbf{0}) \setminus \{e\}$.

Let $E \equiv a.r.b.\mathbf{0}|r.c.\mathbf{0}$ and $F \equiv c.d.\mathbf{0}$. Applying our definition we get that $\mathit{Ref}(r, E, F) \equiv a.c.d.b.\mathbf{0}|c.d.c.\mathbf{0}$. As expected, since in E there are two occurrences of r we replace them with two copies of F. In this way it is possible that new synchronizations are generated. □

From now on when we write $\mathit{Ref}(r, E, F)$ we tacitly assume that r is refinable in E with F.

Notice that we do not allow the use of the parallel composition in the process F. In fact, if $E \equiv r.a.\mathbf{0}$ and $F \equiv b.\mathbf{0}|c.\mathbf{0}$, by applying our notion of refinement we would obtain the process $b.a.\mathbf{0}|c.a.\mathbf{0}$, i.e., we would duplicate part of E. Usually this undesired behavior is avoided by exploiting the concatenation operator ";" in the definition of action refinement (obtaining the process $(b.\mathbf{0}|c.\mathbf{0}); a.\mathbf{0}$). Here instead, we prefer to impose restrictions on F. This assumption is only mildly restrictive since, if F is a finite state[3] process, then there always exists a process F_1, strongly bisimilar to F, which does not contain any occurrence of the parallel operator (see [17]). For instance, in the previous example it is sufficient to consider $F_1 \equiv b.c.\mathbf{0} + c.b.\mathbf{0}$ in order to get the expected $\mathit{Ref}(r, E, F_1) \equiv b.c.a.\mathbf{0} + c.b.a.\mathbf{0}$.

At any fixed abstraction level during the top-down development of a program, it is unrealistic to think that there is just one action to be refined at that level. Compositional properties of the refinement operation allow us to discard the ordering in which the refinements occur.

First we show that our refinement is local to the components in which the action to be refined occurs. This is a consequence of the following theorem.

Theorem 1. *Let $E_1, \ldots, E_n$ and F be terms. Let $C[Z_1, \ldots, Z_n]$ be a context with no occurrences of r and $\bar{r}$. It holds*

$$\mathit{Ref}(r, C[E_1, \ldots, E_n], F) \;\equiv\; C[\mathit{Ref}(r, E_1, F), \ldots, \mathit{Ref}(r, E_n, F)].$$

Hence, if we have a term G which is of the form $E_1|E_2|\ldots|E_n$ and the action r occurs only in E_i it is sufficient to apply the refinement to E_i to obtain $\mathit{Ref}(r, G, F) \equiv E_1|E_2|\ldots|\mathit{Ref}(r, E_i, F)|\ldots|E_n$.

[3] A process is finite state if it reaches only a finite number of different processes. Notice that a finite state process can be recursive.

Example 3. Consider the process $G \equiv recV.(a.V + recW.(a.W + r.W))$. We can decompose it into $C[Z] \equiv recV(a.V + Z)$ and $E \equiv recW.(a.W + r.W)$ and apply the refinement to E. If $F \equiv b.c.\mathbf{0}$ we get that $Ref(r, E, F) \equiv recW.(a.W + b.c.W)$. Hence, $Ref(r, G, F) \equiv recV.(a.V + recW.(a.W + b.c.W))$. □

If we need to refine two actions in a process E, then the order in which we apply the refinements is irrelevant.

Theorem 2. *Let E be a term. Let F_1 and F_2 be two terms with no occurrences of r_1, r_2, $\bar{r}_1$, and $\bar{r}_2$.*

$$Ref(r_2, Ref(r_1, E, F_1), F_2) \equiv Ref(r_1, Ref(r_2, E, F_2), F_1).$$

Example 4. Let $E \equiv r_1.a.\mathbf{0} + r_2.b.r_2.\mathbf{0}$, $F_1 \equiv b.\mathbf{0}$ and $F_2 \equiv c.\mathbf{0}$. We have that $Ref(r_2, Ref(r_1, E, F_1), F_2) \equiv b.a.\mathbf{0} + c.b.c.\mathbf{0} \equiv Ref(r_1, Ref(r_2, E, F_2), F_1)$. □

Moreover, we can refine r_1 in E using F_1 and r_2 in F_1 using F_2 independently from the order in which the refinements are applied as stated by the following theorem.

Theorem 3. *Let E, F_1, F_2 be terms such that r_1 and $\bar{r}_1$ do not occur in F_2.*

$$Ref(r_2, Ref(r_1, E, F_1), F_2) \equiv Ref(r_1, Ref(r_2, E, F_2), Ref(r_2, F_1, F_2)).$$

Example 5. Let $E \equiv r_1.a.\mathbf{0} + a.r_2.\mathbf{0}$, $F_1 \equiv b.r_2\mathbf{0}$ and $F_2 \equiv c.\mathbf{0}$. We have $Ref(r_2, Ref(r_1, E, F_1), F_2) \equiv Ref(r_2, b.r_2.a.\mathbf{0} + a.r_2.\mathbf{0}, F_2) \equiv b.c.a.\mathbf{0} + a.c.\mathbf{0} \equiv Ref(r_1, r_1.a.\mathbf{0} + a.c.\mathbf{0}, b.c.\mathbf{0}) \equiv Ref(r_1, Ref(r_2, E, F_2), Ref(r_2, F_1, F_2))$. □

4 Preserving Process Properties under Refinement

A process property $\mathcal{P}$ is nothing but a class of processes, i.e., the class of processes which satisfy $\mathcal{P}$. In particular, we are interested in classes of processes expressing security notions, i.e., classes of processes which are all secure (with respect to a particular notion of security). We intend to investigate conditions under which notions of security are preserved under action refinement. This correspond to analyze conditions under which classes of processes are closed with respect to action refinement.

We start by characterizing our notion of refinement uniquely in terms of context composition, spelling out the recursion on the structure of E. To do this we introduce a suitable operation which realizes the necessary links from the parts of E which precede an occurrence of r and the parts of E which follow that occurrence. In other words we have to hook F to E, whenever an action r occurs.

We define the set $E@r$ of the parts of E which syntactically follow the outermost occurrences of an action r, and the context $E\{r\}$ which represents the part of E before the outermost occurrences of r.

Definition 6 ($E@r$ and $E\{r\}$). *Let E be a SPA term and r be a refinable action in E. The set of terms $E@r$ is inductively defined as follows:*

$$\begin{array}{ll} \mathbf{0}@r = \emptyset; & Z@r = \emptyset; \\ (r.T)@r = \{T\}; & (a.T)@r = T@r, \ \textit{if } a \neq r; \\ (T_1 + T_2)@r = T_1@r \cup T_2@r; & (T_1|T_2)@r = T_1@r \cup T_2@r; \\ (T \setminus v)@r = T@r; & (T[f])@r = T@r; \\ (recZ.T)@r = T@r. & \end{array}$$

Let $E@r = \{T_1, \ldots, T_n\}$ and $\{X_{T_1}, \ldots, X_{T_n}\}$ be a set of distinct variables indexed in the elements of $E\{r\}$. The context $E\{r\}$ is inductively defined as follows

$$\begin{array}{ll} \mathbf{0}\{r\} = \mathbf{0}; & Z\{r\} = Z; \\ (r.T)\{r\} = X_T; & (a.T)\{r\} = a.(T\{r\}), \ \textit{if } a \neq r; \\ (T_1 + T_2)\{r\} = T_1\{r\} + T_2\{r\}; & (T_1|T_2)\{r\} = T_1\{r\}|T_2\{r\}; \\ (T \setminus v)\{r\} = (T\{r\}) \setminus v; & (T[f])\{r\} = (T\{r\})[f]; \\ (recZ.T)\{r\} = recZ.(T\{r\}). & \end{array}$$

Notice that if $E@r = \{T_1, \ldots, T_n\}$, then $\{X_{T_1}, \ldots, X_{T_n}\}$ is the set of free variables of $E\{r\}$. In the following we will write $E\{r\}$ to denote the context $E\{r\}[X_{T_1}, \ldots, X_{T_n}]$. Thus $E\{r\}[S_1, \ldots, S_n]$ represents the term obtained from $E\{r\}[X_{T_1}, \ldots, X_{T_n}]$ by simultaneously replacing all the free occurrences of the variables $X_{T_1}, \ldots, X_{T_n}$ with the terms $S_1, \ldots, S_n$, respectively.

Example 6.

- Let $E \equiv r.\mathbf{0}|a.\mathbf{0}$. We have that $E@r$ is $\{\mathbf{0}\}$ and $E\{r\}$ is $X_{\mathbf{0}}|a.\mathbf{0}$.
- Let $E \equiv (a.r.\mathbf{0} + b.r.c.r.a.\mathbf{0}) \mid r.\mathbf{0}$. The set $E@r$ contains two processes and is equal to $\{\mathbf{0}, c.r.a.\mathbf{0}\}$. Note that the term $c.r.a.\mathbf{0}$ in $E@r$ contains an occurrence of r. The context $E\{r\}$ is $(a.X_{\mathbf{0}} + b.X_{c.r.a.\mathbf{0}}) \mid X_{\mathbf{0}}$. The set of the free variables of $E\{r\}$ is exactly $\{X_T \mid T \in E@r\}$.
- Let $E \equiv recZ.(a.Z+r.Z)$. We have that $E@r$ is $\{Z\}$ and $E\{r\}$ is $recZ.(a.Z+X_Z)$. In this case $E@r$ has only one element which is not a process. □

The refinement of an action r in E with F can be equivalently obtained by successive context compositions as follows.

Definition 7 (Partial Refinement). *Let E and F be terms, and $r \in Act$ be an action refinable in E with F. Let Y be a variable which does not occur neither in E nor in $E\{r\}$ nor in F. Let $E@r = \{T_1, \ldots, T_n\}$. The* partial refinement *$ParRef(r, E, F)$ of r in E with F is defined as*

$$ParRef(r, E, F) \equiv E\{r\}[F^0[T_1], \ldots, F^0[T_n]].$$

The following theorem provides an alternative characterization of our notion of refinement.

Theorem 4. *The refinement $Ref(r, E, F)$ of r in E with F satisfies*

- *$Ref(r, E, F) \equiv ParRef^0(r, E, F) \equiv E$, if r does not occur in E;*

- $Ref(r, E, F) \equiv ParRef^{n+1}(r, E, F) \equiv ParRef(r, ParRef^n(r, E, F), F)$, *if* r *occurs* $n + 1$ *times in* E.

Intuitively $E@r$ are the parts of E which syntactically follow the occurrences of the action r, while $E\{r\}$ is the part of E which precedes the r's. The holes X_T's in $E\{r\}$ serve to hook the refinement F. Similarly, the free variable Y of $F^0[Y]$ serves to hook the elements of $E@r$ after the execution of F. The partial refinement $ParRef(r, E, F)$ replaces in E as many occurrences as possible of r with F. In the case of nested occurrences of r (e.g., $r.a.r.\mathbf{0}$) the partial refinement replaces only the first occurrence. Hence in order to replace all the occurrences in the worst case it is necessary to compute the partial refinement n times, where n is the number of occurrences of r in E. We would obtain the same result by arbitrarily choosing at each step one occurrence of r replacing it with F, and going on until there are no more occurrences of the refineble action r.

Example 7. We consider again the second process of Example 6, i.e., let $E \equiv (a.r.\mathbf{0} + b.r.c.r.a.\mathbf{0}) \mid r.0$ and $F \equiv e.f.\mathbf{0}$. The partial refinement $ParRef(r, E, F)$ is the process $E' \equiv (a.e.f.\mathbf{0} + b.e.f.c.r.a.\mathbf{0}) \mid e.f.\mathbf{0}$. The context $E'\{r\}$ coincides with $(a.e.f.\mathbf{0} + b.e.f.c.X_{a.\mathbf{0}}) \mid e.f.\mathbf{0}$. Hence $Ref(r, E, F) = ParRef(r, E', F) = (a.e.f.\mathbf{0} + b.e.f.c.e.f.a.\mathbf{0}) \mid e.f.\mathbf{0}$. □

Let $\mathcal{P}$ be a generic process property. We are now ready to introduce some conditions which imply that $\mathcal{P}$ is preserved under action refinement.

Definition 8 ($\mathcal{P}$-refinable contexts). *Let $\mathcal{P}$ be a class of processes. A class $\mathcal{C}$ of contexts is said to be a class of $\mathcal{P}$-*refinable contexts *if:*

- *if $C \in \mathcal{C}$ and C is a process, then $C \in \mathcal{P}$;*
- *if $C, D \in \mathcal{C}$, then $C[D] \in \mathcal{C}$;*
- *if $C \in \mathcal{C}$ and r is refinable in C, then $C@r \cup \{C\{r\}\} \subseteq \mathcal{C}$.*

Theorem 5. *Let $\mathcal{P}$ be a class of processes and $\mathcal{C}$ be a class of $\mathcal{P}$-refinable contexts. Let E and F be processes. If $E, F^0[Y] \in \mathcal{C}$, then $Ref(r, E, F)$ is a process in $\mathcal{P}$ and it is a $\mathcal{P}$-refinable context in $\mathcal{C}$.*

In order to apply Theorem 5 we need to be able to characterize classes of $\mathcal{P}$-refinable contexts. In the following section we analyze one of the security property considered in [3], namely *P_BNDC*, and we show how to apply Theorem 5.

5 Action Refinement and Information Flow Security

Information flow security in a multilevel system aims at guaranteeing that no high level (confidential) information is revealed to users running at low security levels [11, 9, 16], even in the presence of any possible malicious process (attacker). *Persistent Bisimulation-based Non Deducibility on Composition* (*P_BNDC*, for short) [10] is an information flow security property suitable to analyze processes in completely dynamic hostile environments, i.e., environments which can be

dynamically reconfigured at run-time. The notion of P_BNDC is based on the idea of Non-Interference [12] and requires that every state which is reachable by the system still satisfies a basic Non-Interference property. If this holds, one is assured that even if the environment changes during the execution no malicious attacker will be able to compromise the system, as every possible reachable state is guaranteed to be secure. In this paper we present P_BNDC through its unwinding characterization (see [3]).

The definition of P_BNDC in terms of unwinding condition points out that all the high level actions of a P_BNDC process can be locally simulated by a sequence of τ actions.

Definition 9 (P_BNDC). *A process E is P_BNDC if for all E' reachable from E and for all $h \in H$, if $E' \xrightarrow{h} E''$, then $E' \stackrel{\hat{\tau}}{\Longrightarrow} E'''$ with $E'' \setminus H \approx E''' \setminus H$.*

Example 8. Let $l \in L$ and $h \in H$. The process $h.l.h.\mathbf{0} + \tau.l.\mathbf{0}$ is P_BNDC. The process $h.l.\mathbf{0}$ is not P_BNDC. □

Example 9. Let us consider a distributed data base (adapted from [14]) which can take two values and which can be both queried and updated. In particular, the high level user can query it through the high level actions qry_1 and qry_2, while the low level user can only update it through the low level actions upd_1 and upd_2. Hence $qry_1, qry_2 \in H$ and $upd_1, upd_2 \in L$. We can model the data base with the SPA process E defined as

$$\begin{aligned} E \equiv recZ.(&qry_1.Z + upd_1.Z + \tau.Z + \\ &upd_2.recW.(qry_2.W + upd_2.W + \tau.W + upd_1.Z)). \end{aligned}$$

The process E is P_BNDC. Indeed, whenever a high level user queries the data base with a high level action moving the system to a state X then a τ action moving the system to the same state X may be performed, thus masking the high level interactions with the system to low level users. □

The decidability of P_BNDC has been proved in [10] and an efficient (polynomial) algorithm has been presented in [3]. A proof system which allows us to incrementally build P_BNDC processes has been obtained by exploiting both the unwinding characterization of P_BNDC (Definition 9) and the compositional properties of P_BNDC with respect to most of the operators of the SPA language. Here we exploit the same compositionality properties to define classes of P_BNDC and P_BNDC-refinable contexts.

Definition 10 (The classes $\mathcal{C}_{rec}$ and $\mathcal{C}_{par}$).

- *$\mathcal{C}_{rec}$ is the class of contexts containing: the process $\mathbf{0}$; Z, where Z is a variable; $l.C$, with $l \in L \cup \{\tau\}$, $h.C + \tau.C$, with $h \in H$, $C \setminus v$, $C[f]$, $C_1 + C_2$, and $recZ.C$, with $C, C_1, C_2 \in \mathcal{C}_{rec}$.*
- *$\mathcal{C}_{par}$ is the class of contexts containing: the process $\mathbf{0}$; Z, where Z is a variable; $l.C$, with $l \in L \cup \{\tau\}$, $h.C + \tau.C$, with $h \in H$, $C \setminus v$, $C[f]$, $C_1 + C_2$, and $C_1|C_2$, with $C, C_1, C_2 \in \mathcal{C}_{par}$.*

Theorem 6. *The classes $\mathcal{C}_{rec}$ and $\mathcal{C}_{par}$ are* P_BNDC*-refinable.*

Next corollary is an immediate consequence of Theorems 5 and 6.

Corollary 1. *Let E and F be processes. If $E, F^0[Y] \in \mathcal{C}_{rec}$ (resp. $\in \mathcal{C}_{par}$), then $Ref(r, E, F)$ is a P_BNDC process and it is in $\mathcal{C}_{rec}$ (resp. $\in \mathcal{C}_{par}$).*

Example 10. Consider again the abstract specification of the distributed data base represented through the SPA process E of Example 9. The process E belongs to the class $\mathcal{C}_{rec}$ of Definition 10. In fact, $C_1 \equiv qry_2.W + upd_2.W + \tau.W + upd_1.Z \in \mathcal{C}_{rec}$, then $C_2 \equiv recW.C_1 \in \mathcal{C}_{rec}$. Hence, $C_3 \equiv qry_1.Z + upd_1.Z + \tau.Z + upd_2.C_2 \in \mathcal{C}_{rec}$. Thus $E \equiv recZ.C_3 \in \mathcal{C}_{rec}$.

We can refine the update actions by requiring that each update is requested and confirmed, i.e., we refine upd_1 using $F_1 \equiv req_1.cnf_1.\mathbf{0}$ and upd_2 using $F_2 \equiv req_2.cnf_2.\mathbf{0}$, where $req_1, cnf_1, req_2, cnf_2$ are low security level actions. We obtain that the process $Ref(upd_2, Ref(upd_1, E, F_1), F_2)$ is

$$
\begin{aligned}
&recZ.(qry_1.Z + req_1.cnf_1.Z + \tau.Z + \\
&\qquad req_2.cnf_2.recW.(qry_2.W + req_2.cnf_2.W + \tau.W + req_1.cnf_1.Z)).
\end{aligned}
$$

Since $F_1^0[Y]$ and $F_2^0[Y]$ are in C_{rec}, by applying Theorem 1 we have that the process $Ref(upd_2, Ref(upd_1, E, F_1), F_2)$ is *P_BNDC*. □

By exploiting the compositionality of *P_BNDC* with respect to ! (see [4]) we obtain that the above results hold also if we extend the class $\mathcal{C}_{par}$ by including all the contexts of the form $!C$ with $C \in \mathcal{C}_{par}$.

We conclude this section observing that it is immediate to prove Theorem 1 also for the properties *Compositional P_BNDC* (*CP_BNDC*, for short) and *Progressing P_BNDC* (*PP_BNDC*, for short) presented in [3].

6 Related Work

Action refinement has been extensively studied in the literature. There are essentially two interpretations of action refinement: *semantic* and *syntactic* (see [13]). In the semantic interpretation an explicit refinement operator, written $E[r \to F]$, is introduced in the semantic domain used to interpret the terms of the algebra. The semantics of $E[r \to F]$ models the fact that r is an action of E to be refined by process F. In the syntactic approach, the same situation is modelled by syntactically replacing r by F in E. The replacement can be *static*, i.e., before execution, or *dynamic*, i.e., r is replaced as soon as it occurs while executing E. In order to correctly formalize the replacement, the process algebra is usually equipped with an operation of sequential composition (rather than the more standard action prefix), as, e.g., in ACP, since otherwise it would not be closed under the necessary syntactic substitution. Our approach to action refinement follows the static, syntactic interpretation. However, the use of context composition to realize the refinement allows us to keep the original SPA language without introducing a sequential composition operator for processes.

Our definition of action refinement is equivalent, in most cases, to the classical static syntactic approaches presented in the literature. We show this by comparing our definition with the one proposed by Aceto and Hennessy in [1] to model action refinement for CCS processes. First, observe that the language considered in [1] is a variation of CCS with the sequential operator ; but without recursion and renaming. Moreover, their semantics is expressed as a strong bisimilarity extended with a condition on the termination of processes, here denoted by $\sim_{\surd}$. In [1] a refinement is nothing but a function $\rho : \mathcal{L} \rightarrow \mathcal{E}$ which maps each action a into its refinement. Given a process E its refinement $E\rho$ is obtained by syntactically replacing each action a occurring in E with $\rho(a)$. Since by Definition 4 we can avoid the parallel operator in the refining process F, the following theorem holds.

Theorem 7. *Let E and F be two processes without recursion and renaming. Consider the function $\rho : \mathcal{L} \rightarrow \mathcal{E}$ defined as*

$$\rho(a) = \begin{cases} F & \textit{if } a = r \\ a & \textit{otherwise} \end{cases}$$

Let $E\rho$ be the refinement of E with ρ as defined in [1]. If F is a guarded process, then

$$E\rho \sim_{\surd} \mathit{Ref}(r, E, F).$$

Action refinement is also classified as *atomic* or *non-atomic*. Atomic refinement is based on the assumption that actions are atomic and their refinements should in some sense preserve this atomicity (see, e.g.,[7, 5]). As an example, consider the processes $E \equiv r.\mathbf{0}|b.\mathbf{0}$ and $F \equiv a_1.a_2.\mathbf{0}$. The refinement of r in E with F is a process $(a_1.a_2).\mathbf{0}|b.\mathbf{0}$ where the execution of $a_1.a_2.\mathbf{0}$ is non-interruptible, i.e., action b cannot be executed in between the execution of a_1 and a_2. On the other hand, non-atomic refinement is based on the view that atomicity is always relative to the current level of abstraction and may, in a sense, be destroyed by the refinement (see, e.g., [1, 8, 20]). In this paper we follow the *non-atomic* approach. Actually, this approach is on the whole more popular then the former.

In the literature the term *refinement* is also used to indicate any transformation of a system that can be justified because the transformed system implements the original one on the *same* abstraction level, by being more nearly executable, for instance more deterministic. The implementation relation is expressed in terms of pre-orders such as trace inclusion or various kinds of simulation. Many papers in this tradition can be found in [6]. The relations between this form of refinement and information flow security have been studied in [2].

References

1. L. Aceto and M. Hennessy. Adding Action Refinement to a Finite Process Algebra. *Information and Computation*, 115(2):179–247, 1994.
2. A. Bossi, R. Focardi, C. Piazza, and S. Rossi. Refinement Operators and Information Flow Security. In *Proc. of the 1st IEEE Int. Conference on Software Engineering and Formal Methods (SEFM'03)*, pages 44–53. IEEE, 2003.

3. A. Bossi, R. Focardi, C. Piazza, and S. Rossi. Verifying Persistent Security Properties. *Computer Languages, Systems and Structures*, 2004. To appear. Available at `http://www.dsi.unive.it/~srossi/cl04.ps`.
4. A. Bossi, D. Macedonio, C. Piazza, and S. Rossi. Information Flow Security and Recursive Systems. In *Proc. of the Italian Conference on Theoretical Computer Science (ICTCS'03)*, volume 2841 of *LNCS*, pages 369–382. Springer-Verlag, 2003.
5. G. Boudol. Atomic Actions. *Bulletin of the EATCS*, 38:136–144, 1989.
6. J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors. *Stepwise Refinement of Distributed Systems, Models, Formalisms, Correctness, REX Workshop, Mook, The Netherlands, May 29 - June 2, 1989, Proceedings*, volume 430 of *Lecture Notes in Computer Science*. Springer, 1990.
7. J. W. de Bakker and E. P. de Vink. Bisimulation Semantics for Concurrency with Atomicity and Action Refinement. *Fundamenta Informaticae*, 20(1):3–34, 1994.
8. P. Degano and R. Gorrieri. A Causal Operational Semantics of Action Refinement. *Information and Computation*, 122(1):97–119, 1995.
9. R. Focardi and R. Gorrieri. Classification of Security Properties (Part I: Information Flow). In R. Focardi and R. Gorrieri, editors, *Proc. of Foundations of Security Analysis and Design (FOSAD'01)*, volume 2171 of *LNCS*, pages 331–396. Springer-Verlag, 2001.
10. R. Focardi and S. Rossi. Information Flow Security in Dynamic Contexts. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'02)*, pages 307–319. IEEE, 2002.
11. S. N. Foley. A Universal Theory of Information Flow. In *Proc. of the IEEE Symposium on Security and Privacy (SSP'87)*, pages 116–122. IEEE, 1987.
12. J. A. Goguen and J. Meseguer. Security Policies and Security Models. In *Proc. of the IEEE Symposium on Security and Privacy (SSP'82)*, pages 11–20. IEEE, 1982.
13. U. Goltz, R. Gorrieri, and A. Rensink. Comparing Syntactic and Semantic Action Refinement. *Information and Computation*, 125(2):118–143, 1996.
14. R. Gorrieri and A. Rensink. Action Refinement. Technical Report UBLCS-99-09, University of Bologna (Italy), 1999.
15. H. Mantel. Possibilistic Definitions of Security - An Assembly Kit -. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'00)*, pages 185–199. IEEE, 2000.
16. J. McLean. Security Models and Information Flow. In *Proc. of the IEEE Symposium on Security and Privacy (SSP'90)*, pages 180–187. IEEE, 1990.
17. J. K. Millen. Unwinding Forward Correctability. In *Proc. of the IEEE Computer Security Foundations Workshop (CSFW'94)*, pages 2–10. IEEE, 1994.
18. R. Milner. *Communication and Concurrency.* Prentice-Hall, 1989.
19. M. Nielsen, U. Engberg, and K. S. Larsen. Fully Abstract Models for a Process Language with Refinement. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Workshop on Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, volume 354 of *LNCS*, pages 523–548. Springer-Verlag, 1989.
20. R. J. van Glabbeek and U. Goltz. Refinement of Actions and Equivalence Notions for Concurrent Systems. *Acta Informatica*, 37(4/5):229–327, 2001.
21. N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, 1971.

Totally Correct Logic Program Transformations Using Well-Founded Annotations

Alberto Pettorossi[1], Maurizio Proietti[2]

(1) DISP, University of Tor Vergata, Roma, Italy. pettorossi@info.uniroma2.it
(2) IASI-CNR, Roma, Italy. proietti@iasi.rm.cnr.it

(Extended Abstract)

Program transformation is one of the most prominent methodologies for the development of declarative programs and, in particular, functional and logic programs [2,5,9]. The main advantage of this methodology is that it allows one to deal with the issue of program correctness and the issue of program efficiency in a separated manner. One first writes a simple, maybe inefficient, program whose correctness can easily be proved, and then one derives a more efficient program by applying some given transformation rules which preserve program correctness.

In the case of definite logic programs, which are of our interest here, the correctness of the initial program is often very easy to prove because, usually, it is very close to the formal specification of that same program. On the contrary, the proof that the rules preserve program correctness is often more intricate (as it is also the case for functional programs). In particular, these correctness proofs cannot be done *in isolation*, in the sense that the correctness of a single transformation rule depends, in general, on the other rules one applies for transforming programs.

The correctness of the rules can be either partial or total. We say that a rule which transforms program P_1 into program P_2 is *partially correct* iff $M(P_1) \supseteq M(P_2)$, where $M(P)$ denotes the least Herbrand model of any given program P. Analogously, we say that a rule which transforms program P_1 into program P_2 is *totally correct* iff $M(P_1) = M(P_2)$.

Partial correctness is a straightforward consequence of the fact that the transformation rules, and in particular the familiar *unfold/fold* rules, basically consist in applying logical equivalences [9]. Indeed, whenever we derive a program P_2 from a program P_1 by replacing a formula A by a formula B such that $M(P_1) \models A \leftrightarrow B$, we get $M(P_1) \supseteq M(P_2)$. However, it is well known that the opposite inclusion $M(P_1) \subseteq M(P_2)$ may not hold and, thus, in general, the unfold/fold transformations are not totally correct as shown by the following simple example. Let us consider the transformation of program P_1 into program P_2, where P_1 and P_2 are as follows:

P_1: $p \leftarrow q$
$\quad\quad q \leftarrow$

P_2: $p \leftarrow p$
$\quad\quad q \leftarrow$

This transformation, which corresponds to an application of the *folding* rule, is justified by the fact that the equivalence $M(P_1) \models p \leftrightarrow q$ holds. However, the

least Herbrand model is not preserved because we have that $M(P_1) = \{p, q\} \supset \{q\} = M(P_2)$.

In the case of non-propositional programs it is not easy to check whether or not the application of an unfold/fold transformation rule is totally correct (actually, it can be shown that this is an undecidable problem). For this reason, in their landmark paper Tamaki and Sato proposed suitable applicability conditions which ensure the total correctness of the transformations [9]. These conditions are based on: (i) the form of the clauses that can be used in a folding step, and (ii) annotations of the program clauses that depend on the transformation history, that is, on the sequence of transformation rules applied during a program derivation. In particular, they stipulate that: (i) one is allowed to fold a clause by using a non-recursive clause which is marked as *'foldable'*, and (ii) a clause is marked as *'foldable'* if it is derived by unfolding. Thus, conditions (i) and (ii) express that a clause can be folded only if it is derived by unfolding at a previous transformation step.

Tamaki-Sato's approach has been extended in several papers (see, for instance, [4,6,8,10]) by: (i) relaxing the restrictions on the clauses that can be used in a folding step, and (ii) generalizing the history dependent program annotations. The most recent of these papers [8] presents sufficient conditions for the total correctness of the unfold/fold transformations in the case where several, possibly recursive clauses are used in a folding step. These conditions are based on some measures which are incremented or decremented when the unfolding or folding rules are applied.

Unfortunately, the proofs of total correctness of the unfold/fold transformations presented in [4,6,8,9,10], use rather complex, *ad hoc* techniques, and it is very difficult to understand why they work and how they could be generalized for dealing with other program transformations or language extensions.

The main contribution of this paper is a logical foundation of the theory of total correctness of logic program transformations (and in particular unfold/fold transformations). Our theory is based on the notion of *well-founded annotations* and the *unique fixpoint principle.*

A well-founded annotation is a mapping α that associates with every clause $H \leftarrow A_1 \wedge \ldots \wedge A_k$ of a program P an annotated clause of the form:

$$H\{N\} \leftarrow c(N, N_1, \ldots, N_k) \wedge A_1\{N_1\} \wedge \ldots \wedge A_k\{N_k\}$$

where: (i) the annotation variables $N, N_1, \ldots, N_k$ range over a set W and should be considered as extra arguments of the atoms occurring in the clause, and (ii) for $i = 1, \ldots, k$, the relation $c(N, N_1, \ldots, N_k)$ implies $N > N_i$, where $>$ is a well-founded ordering on W. By applying the well-founded annotation α to every clause in P, we get an annotated program $\alpha(P)$ that, by construction, enjoys the following two properties: (1) for every ground atom A, $A \in M(P)$ iff there exists $n \in W$ such that $A\{n\} \in M(\alpha(P))$, and (2) for every ground annotated atom $A\{n\}$, $\alpha(P) \cup \{\leftarrow A\{n\}\}$ has a finite SLD tree, that is, $\alpha(P)$ is *terminating.* By Property (2), the least Herbrand model of $\alpha(P)$ is the *unique fixpoint* of the immediate consequence operator $T_{\alpha(P)}$ [1].

Based on well-founded annotations, we propose a method for totally correct transformations of definite logic programs. Given a program P_1 our method allows us to derive a program P_2 by the following steps: (i) we choose a well-founded annotation α_1 so that from program P_1 we produce an annotated program $\alpha_1(P_1)$, (ii) we apply suitable variants of the unfold/fold rules for transforming annotated programs so that from $\alpha_1(P_1)$ we derive a new *terminating* annotated program $\alpha_2(P_2)$, with α_2 possibly different from α_1, and finally, (iii) from $\alpha_2(P_2)$ we get program P_2 by erasing the annotations. The fact that $\alpha_2(P_2)$ is terminating is enforced by the transformation rules because they preserve the well-founded ordering $>$, in the sense that, for every clause derived by applying the rules, the annotation of the head is greater (w.r.t. ¿) than the annotation of every atom in the body.

The total correctness of the transformation, that is, $M(P_1) = M(P_2)$, is proved as follows. On one hand, the transformation rules act on non-annotated clauses like the usual unfold/fold rules and, as already mentioned, they ensure partial correctness, that is, $M(P_1) \supseteq M(P_2)$. On the other hand, since $\alpha_2(P_2)$ is terminating, by the unique fixpoint principle [3,7] we have that $M(\alpha_1(P_1)) \subseteq M(\alpha_2(P_2))$ and, thus, by Property (1) of well-founded annotations, $M(P_1) \subseteq M(P_2)$.

Notice that in our method neither P_1 nor P_2 is required to be terminating. Moreover, our method is parametric w.r.t. the well-founded annotations and, in particular, w.r.t. the well-founded ordering $>$ used for the derivation of P_2 from P_1. By suitable choices of this ordering we can prove the total correctness of the various variants of the unfold/fold rules proposed in the literature [4,6,8,9,10].

An Example

We revisit an example of program transformation taken from [8] where the total correctness proof is rather intricate. We show that, on the contrary, the total correctness of this transformation can easily be established by our well-founded annotation method. Let us consider the following program P_1:

1. $thm(X) \leftarrow gen(X) \wedge test(X)$
2. $gen([\,]) \leftarrow$
3. $gen([0|X]) \leftarrow gen(X)$
4. $test(X) \leftarrow canon(X)$
5. $test(X) \leftarrow trans(X,Y) \wedge test(Y)$
6. $canon([\,]) \leftarrow$
7. $canon([1|X]) \leftarrow canon(X)$
8. $trans([0|X],[1|X]) \leftarrow$
9. $trans([1|X],[1|Y]) \leftarrow trans(X,Y)$

where $thm(X)$ holds iff X is a string of 0's that can be transformed into a string of 1's by repeated applications of $trans(X,Y)$. Given the string X, the predicate $trans(X,Y)$ generates the string Y by replacing the leftmost 0 in X by 1. Let us consider the well-founded annotation α_1 that associates with every clause:

$H \leftarrow A_1 \wedge \ldots \wedge A_k$

the annotated clause:

$H\{N\} \leftarrow N \geq N_1 + \ldots + N_k + 1 \wedge A_1\{N_1\} \wedge \ldots \wedge A_k\{N_k\}$

where the annotation variables $N, N_1, \ldots, N_k$ range over non-negative integers and $\geq$ is the usual 'greater or equal' ordering over integers. Thus, the annotated program $\alpha_1(P_1)$ is the following one:

1a. $\mathit{thm}(X)\{N\} \leftarrow N \geq N_1 + N_2 + 1 \wedge \mathit{gen}(X)\{N_1\} \wedge \mathit{test}(X)\{N_2\}$
2a. $\mathit{gen}([\,])\{0\} \leftarrow$
3a. $\mathit{gen}([0|X])\{N\} \leftarrow N \geq N_1 + 1 \wedge \mathit{gen}(X)\{N_1\}$
4a. $\mathit{test}(X)\{N\} \leftarrow N \geq N_1 + 1 \wedge \mathit{canon}(X)\{N_1\}$
5a. $\mathit{test}(X)\{N\} \leftarrow N \geq N_1 + N_2 + 1 \wedge \mathit{trans}(X, Y)\{N_1\} \wedge \mathit{test}(Y)\{N_2\}$
6a. $\mathit{canon}([\,])\{0\} \leftarrow$
7a. $\mathit{canon}([1|X])\{N\} \leftarrow N \geq N_1 + 1 \wedge \mathit{canon}(X)\{N_1\}$
8a. $\mathit{trans}([0|X], [1|X])\{0\} \leftarrow$
9a. $\mathit{trans}([1|X], [1|Y])\{N\} \leftarrow N \geq N_1 + 1 \wedge \mathit{trans}(X, Y)\{N_1\}$

As already mentioned, the annotated program $\alpha_1(P_1)$ can be considered as a logic program where the annotation variables are taken as extra arguments. The annotated program $\alpha_1(P_1)$ is terminating because $N \geq N_1 + \ldots + N_k + 1$ implies that, for $i = 1, \ldots, k$, $N > N_i$. (Also P_1 is terminating, but we need not use this property.) Now, let us construct a totally correct transformation by using unfold/fold transformation rules for annotated programs. The unfolding and folding rules for annotated programs work exactly like the rules for non-annotated programs, by considering the annotation variables as extra arguments. By applying several times the unfolding rule, from clause 1a we derive:

10a. $\mathit{thm}([\,])\{N\} \leftarrow N \geq 2$
11a. $\mathit{thm}([0|X])\{N\} \leftarrow N \geq N_1 + N_2 + 5 \wedge \mathit{gen}(X)\{N_1\} \wedge \mathit{canon}(X)\{N_2\}$
12a. $\mathit{thm}([0|X])\{N\} \leftarrow N \geq N_1 + N_2 + N_3 + 5 \wedge \mathit{gen}(X)\{N_1\} \wedge$
$\qquad \mathit{trans}(X, Y)\{N_2\} \wedge \mathit{test}([1|Y])\{N_3\}$

Now we apply the goal replacement rule and we replace the annotated atom $\mathit{test}([1|Y])\{N_3\}$ by $N_3 \geq N_4 \wedge \mathit{test}(Y)\{N_4\}$. This replacement is justified by the following two properties: (1) $M(P_1) \models \forall Y\,(\mathit{test}([1|Y]) \leftrightarrow \mathit{test}(Y))$ and (2) $M(\alpha_1(P_1)) \models \forall Y \forall N_3\,(\mathit{test}([1|Y])\{N_3\} \rightarrow \exists N_4 (N_3 \geq N_4 \wedge \mathit{test}(Y)\{N_4\}))$. By applying the goal replacement rule, clause 12a is replaced by the following clause:

13a. $\mathit{thm}([0|X])\{N\} \leftarrow N \geq N_1 + N_2 + N_4 + 5 \wedge \mathit{gen}(X)\{N_1\} \wedge$
$\qquad \mathit{trans}(X, Y)\{N_2\} \wedge \mathit{test}(Y)\{N_4\}$

By folding clauses 11a and 13a using clauses 4a and 5a we get:

14a. $\mathit{thm}([0|X])\{N\} \leftarrow N \geq N_1 + N_5 + 4 \wedge \mathit{gen}(X)\{N_1\} \wedge \mathit{test}(X)\{N_5\}$

Finally, by folding clause 14a using clause 1a, we derive:

15a. $\mathit{thm}([0|X])\{N\} \leftarrow N \geq N_6 + 3 \wedge \mathit{thm}(X)\{N_6\}$

The final annotated program is $\alpha_2(P_2) = (\alpha_1(P_1) - \{1a\}) \cup \{10a, 15a\}$. Notice that in clause 15a the annotation of the head is greater than the annotation of the body atom (because $N \geq N_6 + 3$ implies $N > N_6$). Thus, $\alpha_2(P_2)$ is terminating and $M(\alpha_2(P_2))$ is the unique fixpoint of $T_{\alpha_2(P_2)}$. By the unique fixpoint principle [3,7], we deduce that $M(\alpha_1(P_1)) \subseteq M(\alpha_2(P_2))$.

Now, let us consider the program P_2 obtained by dropping the annotations from $\alpha_2(P_2)$, that is, $P_2 = (P_1 - \{1\}) \cup \{10, 15\}$, where clauses 10 and 15 are the following:

10. $thm([\,]) \leftarrow$
15. $thm([0|X]) \leftarrow thm(X)$

Notice that, for every ground atom A, we have that $A \in M(P_1)$ iff there exists a non-negative integer n such that $A\{n\} \in M(\alpha_1(P_1))$, and similarly, $A \in M(P_2)$ iff there exists a non-negative integer n such that $A\{n\} \in M(\alpha_2(P_2))$. Therefore, from $M(\alpha_1(P_1)) \subseteq M(\alpha_2(P_2))$ it follows that $M(P_1) \subseteq M(P_2)$. Since, as already mentioned, the transformation rules act on non-annotated programs like the usual unfold/fold transformations, and these transformations are partially correct, we also have $M(P_1) \supseteq M(P_2)$. Thus, the transformation of P_1 into P_2 is totally correct.

References

1. M. Bezem. Characterizing termination of logic programs with level mappings. In E.L. Lusk and R.A. Overbeek, editors, *Proceedings of the North American Conference on Logic Programming, Cleveland, Ohio (USA)*, pages 69–80. MIT Press, 1989.
2. R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
3. B. Courcelle. Infinite trees in normal form and recursive equations having a unique solution. *Mathematical Systems Theory*, 13:131–180, 1979.
4. M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. In M. Hermenegildo and J. Penjam, editors, *Proceedings Sixth International Symposium on Programming Language Implementation and Logic Programming (PLILP '94)*, Lecture Notes in Computer Science 844, pages 340–354. Springer-Verlag, 1994.
5. C. J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, 1981.
6. T. Kanamori and H. Fujita. Unfold/fold transformation of logic programs with counters. Technical Report 179, ICOT, Tokyo, Japan, 1986.
7. M. Proietti and A. Pettorossi. Transforming inductive definitions. In D. De Schreye, editor, *Proceedings of the 1999 International Conference on Logic Programming*, pages 486–499. MIT Press, 1999.
8. A. Roychoudhury, K. Narayan Kumar, C.R. Ramakrishnan, and I.V. Ramakrishnan. An unfold/fold tranformation framework for definite logic programs. *ACM Transactions on Programming Languages and Systems*, 26:264–509, 2004.
9. H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. In S.-Å. Tärnlund, editor, *Proceedings of the Second International Conference on Logic Programming*, pages 127–138, Uppsala, Sweden, 1984. Uppsala University.
10. H. Tamaki and T. Sato. A generalized correctness proof of the unfold/fold logic program transformation. Technical Report 86-4, Ibaraki University, Japan, 1986.

Implementing Joint Fixpoint Semantics on Top of DLV*

Francesco Buccafurri and Gianluca Caminiti

DIMET, Università di Reggio Calabria, I-89100 Reggio Calabria, Italy,
bucca@unirc.it, caminiti@ing.unirc.it

Abstract. In this paper we propose an implementation of the joint fixpoint semantics on top of the DLV system. Our framework provides a front-end to compromise logic programs (COLPs) representing agents' requirements or desires in a multi-agent environment. By exploiting a direct mapping from COLP programs to classic logic programs we compute COLPs' joint fixpoints modeling a common agreement among agents. Moreover an option is provided for computing minimal joint fixpoints in order to deal with situations where minimality is an issue to be considered.

1 A Framework for Joint Fixpoint Semantics

In a multi-agent environment it is possible to represent agents' requirements or desires as logic programs. Then a suitable semantics like the joint fixpoint (JFP) semantics [1] can be used in order to model any joint decision reflecting a common agreement among such agents. Consider the following example: the members of a family (Dad, Mom and their son Charlie, viewed as three agents) are discussing about buying a new car. Each of them proposes desired or necessary features the car should have.
Dad is more concerned on safety and fuel consumption: he requires twin airbag and ABS (Anti-lock Brake System) and desires to buy (if possible) a city car. He accepts to choose a high displacement car but only if it has a diesel engine. In this case he also accepts to pay for the air conditioner: a low displacement car wouldn't have the required power. Metalized paint is tolerated. Further accessories (e.g. automatic shift, power steering) are considered a waste of money.
Mom is not too much safe with driving so she wants twin airbag in case of an accident. For the same reason she would like a city car otherwise she desires power steering to help her while parking. Moreover, she likes comfort: air conditioner, automatic shift, and compact disk player are fine. High displacement, ABS and metalized paint are accepted, but not really necessary.
Charlie desires as many accessories as possible: an high displacement car with twin airbag (possibly plus ABS), air conditioner, automatic shift is welcome.

* This work was partially supported by WASP (Working Group on Answer Set Programming) IST-2001-37004, 5th framework programme, Information Society Technologies, Action Line FET (Future and emerging technologies).

Moreover, he desires metalized paint together with the CD player, but if the latter is absent then the paint must be metalized. Power steering is accepted (but not needed) and no problem occurs in case of a city car. Since he pays the fuel he consumes, if the displacement is high then a diesel engine would be fine. It is possible to represent the above by three compromise logic programs (COLPs), P_{Dad}, P_{Mom} and $P_{Charlie}$, each expressing the desires and agreements of a different family member:

$$P_{Dad}:$$

$$\begin{array}{rl}
\texttt{twin_airbag} \leftarrow & \\
\texttt{abs_system} \leftarrow & \\
\mathit{okay}(\texttt{city}) \leftarrow & \\
\mathit{okay}(\texttt{high_disp}) \leftarrow & \texttt{diesel} \\
\mathit{okay}(\texttt{air_cond}) \leftarrow & \texttt{high_disp} \\
\mathit{okay}(\texttt{metal}) \leftarrow & \\
\mathit{okay}(\texttt{cd}) \leftarrow &
\end{array}$$

$$P_{Mom}:$$

$$\begin{array}{rl}
\texttt{twin_airbag} \leftarrow & \\
\mathit{okay}(\texttt{air_cond}) \leftarrow & \\
\mathit{okay}(\texttt{auto_shift}) \leftarrow & \\
\mathit{okay}(\texttt{city}) \leftarrow & \\
\texttt{steering} \leftarrow & \mathit{not}\ \texttt{city} \\
\mathit{okay}(\texttt{abs_system}) \leftarrow & \\
\mathit{okay}(\texttt{metal}) \leftarrow & \\
\mathit{okay}(\texttt{cd}) \leftarrow & \\
\mathit{okay}(\texttt{high_disp}) \leftarrow &
\end{array}$$

$$P_{Charlie}:$$

$$\begin{array}{rl}
\mathit{okay}(\texttt{air_cond}) \leftarrow & \\
\mathit{okay}(\texttt{auto_shift}) \leftarrow & \\
\mathit{okay}(\texttt{high_disp}) \leftarrow & \\
\mathit{okay_group}(\texttt{metal,cd}) \leftarrow & \\
\texttt{metal} \leftarrow & \mathit{not}\ \texttt{cd} \\
\mathit{okay}(\texttt{city}) \leftarrow & \\
\mathit{okay}(\texttt{steering}) \leftarrow & \\
\mathit{okay}(\texttt{diesel}) \leftarrow & \texttt{high_disp} \\
\mathit{okay}(\texttt{twin_airbag}) \leftarrow & \\
\mathit{okay}(\texttt{abs_system}) \leftarrow & \texttt{twin_airbag}
\end{array}$$

Furthermore the knowledge about each user includes rules which characterize the fuel type (either petrol or diesel):

$$\begin{array}{l}
\texttt{petrol} \leftarrow \mathit{not}\ \texttt{diesel} \\
\texttt{diesel} \leftarrow \mathit{not}\ \texttt{petrol}
\end{array}$$

A COLP program is based on a language enriched with special predicates [1] like *okay* (resp. *okay_group*) representing single atoms (resp. groups of atoms) which are tolerated, but not required. In particular $\mathit{okay_group}(p_1, \cdots, p_n)$ expresses that the group of arguments $p_1, \cdots, p_n$ is tolerated without implying that $p_1, \cdots, p_n$ are separately tolerated. In a COLP program required atoms are represented by simple facts while refused ones can be simply omitted or explicitly excluded by integrity constraints. For example, agent *Dad* refuses *automatic shift* since this accessory doesn't occur either as a fact or as an argument inside

an *okay* predicate, however this refuse could be also expressed by the rule:

$$\bot \leftarrow \texttt{auto_shift}$$

With regard to the example a common agreement will be reached on any of the following JFPs:

```
{abs_system, twin_airbag, cd, city, metal, petrol}
{abs_system, twin_airbag, air_cond, cd, city, diesel, high_disp, metal}
{abs_system, twin_airbag, cd, city, diesel, metal}
{abs_system, twin_airbag, cd, city, diesel, high_disp, metal}
{abs_system, twin_airbag, city, metal, petrol}
{abs_system, twin_airbag, air_cond, city, diesel, high_disp, metal}
{abs_system, twin_airbag, city, diesel, high_disp, metal}
{abs_system, twin_airbag, city, diesel, metal}
```

Thus, each joint fixpoint represents a possible choice which is accepted by all the agents. Further constraints could be added in order to meet a particular target, e.g. when the car price is expressed as a function of the number and type of chosen accessories and the agents have an upper bound on the money they can pay some of the above fixpoints may be rejected. However we do not consider this possibility in this paper, leaving it as a matter for future work. An interesting case is when agents' default desire is saving money. Here minimal joint fixpoints can be adopted as a solution since they include only the minimum number of atoms on which there is a common agreement. With regard to the example, the MJFPs characterizing the (possibly) cheapest cars are the following:

```
{abs_system, twin_airbag, city, metal, petrol}
{abs_system, twin_airbag, city, diesel, metal}
```

In this work we present a framework which is able to compute either JFPs or MJFPs given an arbitrary set of COLP programs. In particular we perform the translation from COLP programs to classic logic programs and implement the mapping from JFP semantics to SM semantics proposed in [1] in such a way that a single classic logic program is generated whose stable models are in one-to-one correspondence with the JFPs. Moreover our framework enhances the above mapping in order to work with non propositional programs and includes a wrapper exploiting the DLV system to compute the stable models. Finally, we compute the minimal joint fixpoints as a further option.

The rest of the paper is structured as follows: in the next section we give a brief description of JFP semantics and then we consider the mapping from JFP semantics to SM semantics. In Section 3 we describe in detail the sequence of operations implementing the mapping and the algorithms exploited by the framework software modules. Moreover we consider some implementation issues. Finally, we draw in Section 4 our conclusions by considering possible alternative solutions to the mentioned problems and proposing some optimizations to the framework.

2 The Joint Fixpoint Semantics

Before introducing JFP semantics, we briefly recall some basic concepts about logic programming [2, 3] and stable models [5, 6]. Further details about JFP semantics can be found in [1].

2.1 Basic Definitions

A *term* is either a variable or a constant. Variables are denoted by strings starting with uppercase letters, while those starting with lower case letters denote constants. An *atom* or *positive literal* is an expression $p(t_1, \cdots, t_n)$, where p is a *predicate* of arity n and $t_1, \cdots, t_n$ are terms. A *negative literal* is the *negation as failure (NAF) not* a of a given atom a. A *clause* or *rule* r is a formula

$$a \leftarrow b_1 \wedge \cdots \wedge b_k \wedge not\ b_{k+1} \wedge \cdots \wedge not\ b_m \qquad m \geq 0.$$

where $a, b_1, \cdots, b_k$ are positive literals and $not\ b_{k+1}, \cdots, not\ b_m$ are negative literals. a is called the *head* of r, while the conjunction $b_1 \wedge \cdots \wedge b_k \wedge not\ b_{k+1} \wedge \cdots \wedge not\ b_m$ is the *body* of r. When $m = 0$ the rule r is said a *fact*, while if $a = \bot$ then r is said an *integrity constraint*.

A *logic program* is a finite set of rules. A term (an atom, a rule, a program, etc.) is called *ground*, if no variable appears in it. A ground program is also called a *propositional* program.

Let $Var(\mathcal{P})$ be a finite set of atoms. A propositional *logic program* $\mathcal{P}$ defined on $Var(\mathcal{P})$ consists of a finite set of rules whose atoms are all in $Var(\mathcal{P})$.

A logic program is *positive* if no negative literal occurs in it.

An *(Herbrand) interpretation* for a program $\mathcal{P}$ is a subset of $Var(\mathcal{P})$. A positive literal a (resp. a negative literal *not* a) is *true* w.r.t. an interpretation I if $a \in I$ (resp. $a \notin I$); otherwise it is *false*. A rule is *satisfied* (or is *true*) w.r.t. I if its head is true or its body is false w.r.t. I. An interpretation I is a *(Herbrand) model* of a program $\mathcal{P}$ if it satisfies all rules in $\mathcal{P}$.

For each program $\mathcal{P}$, the *immediate consequence operator* $T_{\mathcal{P}}$ is a function from $2^{Var(\mathcal{P})}$ to $2^{Var(\mathcal{P})}$ defined as follows. For each interpretation $I \subseteq Var(\mathcal{P})$, $T_{\mathcal{P}}(I)$ consists of the set of all heads of rules in $\mathcal{P}$ whose bodies evaluate to true in I.

An interpretation I is a *fixpoint* of a logic program $\mathcal{P}$ if I is a fixpoint of the associated transformation $T_{\mathcal{P}}$, i.e., if $T_{\mathcal{P}}(I) = I$. The set of all fixpoints of $\mathcal{P}$ is denoted by $FP(\mathcal{P})$.

Let I be an interpretation of $\mathcal{P}$ and let $a \in Var(\mathcal{P})$ be an atom. We say that a is *supported* by I (in $\mathcal{P}$) if there is a rule of $\mathcal{P}$ with head a whose body evaluates to true in I, i.e., if $a \in T_{\mathcal{P}}(I)$. From the definition of fixpoint it immediately follows that an interpretation I of $\mathcal{P}$ is a fixpoint of $\mathcal{P}$ iff I coincides with the set of all atoms supported by I.

For any interpretation $I \subseteq Var(\mathcal{P})$, we define $T^0_{\mathcal{P}}(I) = I$ and for all $i \geq 0$, $T^{i+1}_{\mathcal{P}}(I) = T_{\mathcal{P}}(T^i_{\mathcal{P}}(I))$. If $\mathcal{P}$ is a positive program, then $T_{\mathcal{P}}$ is monotonic and thus has a least fixpoint $lfp(\mathcal{P}) = T^{\infty}_{\mathcal{P}}(\emptyset)$. This least fixpoint coincides with the least

Herbrand model $lm(\mathcal{P})$ of $\mathcal{P}$, i.e. $lm(\mathcal{P}) = lfp(\mathcal{P})$. For non-positive programs $\mathcal{P}$, $T_{\mathcal{P}}$ isn't in general monotonic, and $\mathcal{P}$ does not necessarily have a least fixpoint (it may even have no fixpoint at all).

2.2 Stable Models

Let $\mathcal{P}$ be a logic program and $I \subseteq Var(\mathcal{P})$ be an interpretation. The *Gelfond-Lifschitz transformation* (or simply *GL-transformation*) of $\mathcal{P}$ w.r.t. I, denoted by $\mathcal{P}^I$ is the program obtained by $\mathcal{P}$ by removing all rules containing a negative literal *not* b in the body such that $b \in I$, and by removing all negative literals from the remaining rules.

Definition 1 ([5]). *Given a logic program $\mathcal{P}$ and an interpretation $M \subseteq Var(\mathcal{P})$, M is a* stable model *of $\mathcal{P}$ if $M = T^{\infty}_{\mathcal{P}^M}(\emptyset)$.*

A logic program $\mathcal{P}$ admits in general a number (possibly zero) of stable models. We denote by $SM(\mathcal{P})$ the set of all stable models of the program $\mathcal{P}$.

2.3 Joint Fixpoints

In this section we recall the Joint Fixpoint Semantics for logic programs [1].

Let $S = \{\mathcal{P}_1, \mathcal{P}_2, \cdots \mathcal{P}_n\}$ be a set of logic programs such that $Var(\mathcal{P}_1) = Var(\mathcal{P}_2) = \cdots = Var(\mathcal{P}_n) = Var$. We define the set $JFP(\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_n)$ of *joint fixpoints* by:

$$JFP(\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_n) \;=\; FP(\mathcal{P}_1) \cap FP(\mathcal{P}_2) \cap \cdots \cap FP(\mathcal{P}_n).$$

In words, $JFP(\mathcal{P}_1, \mathcal{P}_2, \cdots, \mathcal{P}_n)$ consists of all common fixpoints to the programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$.

Moreover, we define the set $MJFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ of *minimal joint fixpoint* as:

$$MJFP(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \{F \in JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n) \mid \nexists F' \in JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n) \wedge F' \subset F\}.$$

$MJFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ consists of all minimal common fixpoints to the programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$.

2.4 Mapping Joint Fixpoints on Stable Models

In this section we briefly recall the translation from Logic Programming under the Joint Fixpoint Semantics to Logic Programming under Stable Model Semantics. Further details can be found in [1].

Definition 2. *Let $\mathcal{P}$ be a program and let M be a set of atoms in $Var(\mathcal{P})$. We denote by $[M]_{\mathcal{P}}$ the set $\{a_{\mathcal{P}} \mid a \in M\} \cup \{a'_{\mathcal{P}} \mid a \in Var(\mathcal{P}) \setminus M\} \cup \{sa_{\mathcal{P}} \mid a \in M\}$.*

Here, with a little abuse of notation, $a_{\mathcal{P}}$, $a'_{\mathcal{P}}$ and $sa_{\mathcal{P}}$ denote new atoms obtained through a sort of renaming operation performed on a, i.e. given an atom a, a *new* atom $sa_{\mathcal{P}}$ is created whose name is the same of a plus a prefix s and a suffix $_{\mathcal{P}}$.

Definition 3. *Let $\mathcal{P}$ be a positive program. We define the program $\Gamma(\mathcal{P})$ over the set of atoms $Var(\Gamma(\mathcal{P})) = \{a_{\mathcal{P}} \mid a \in Var(\mathcal{P})\} \cup \{a'_{\mathcal{P}} \mid a \in Var(\mathcal{P})\} \cup \{sa_{\mathcal{P}} \mid a \in Var(\mathcal{P})\} \cup \{fail_{\mathcal{P}}\}$ as the union of the sets of rules S_1, S_2 and S_3, defined as follows:*

$$S_1 = \{a_{\mathcal{P}} \leftarrow not\ a'_{\mathcal{P}} \mid a \in Var(\mathcal{P})\} \cup \{a'_{\mathcal{P}} \leftarrow not\ a_{\mathcal{P}} \mid a \in Var(\mathcal{P})\}$$

$$S_2 = \{sa_{\mathcal{P}} \leftarrow b^1_{\mathcal{P}}, ..., b^n_{\mathcal{P}} \mid a \leftarrow b_1, \cdots b_n \in \mathcal{P}\}$$

$$S_3 = \{fail_{\mathcal{P}} \leftarrow not\ fail_{\mathcal{P}}, sa_{\mathcal{P}}, not\ a_{\mathcal{P}} \mid a \in Var(\mathcal{P})\} \cup \{fail_{\mathcal{P}} \leftarrow not\ fail_{\mathcal{P}}, a_{\mathcal{P}}, not\ sa_{\mathcal{P}} \mid a \in Var(\mathcal{P})\}.$$

The rules included in S_1 guess potential fixpoints among the atoms which are in $Var(\mathcal{P})$. Given an atom $a \in Var(\mathcal{P})$, $a_{\mathcal{P}}$ represents a possible element of a fixpoint of $\mathcal{P}$, while $a'_{\mathcal{P}}$ is its negated version so that only one of them can be part of the same fixpoint. In S_2 there are rules which characterize atoms which are possibly included in stable models, and finally the rules in S_3 state that an atom a is part of a fixpoint iff it is included in a stable model, i.e. fixpoints must be also stable models and vice-versa.

In [1] it was shown that a one-to-one correspondence exists between the set of fixpoints of a given program $\mathcal{P}$ and the set $SM(\Gamma(\mathcal{P}))$ of stable models of the program $\Gamma(\mathcal{P})$.

Now suppose we have a set of positive programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$ over the same set of propositional variables. In [1] it was found a program $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ associated to the set of programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$ such that the stable models of $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ correspond to the joint fixpoints of $\mathcal{P}_1, \cdots, \mathcal{P}_n$. $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is constructed by performing the union of all the programs $\Gamma(\mathcal{P}_i)$, for $1 \leq i \leq n$, with another program $C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ that we next define. Informally, under stable model semantics, rules of programs $\Gamma(\mathcal{P}_1)$, $\Gamma(\mathcal{P}_2)$, ... , $\Gamma(\mathcal{P}_n)$ have the effect of generating all the fixpoints of $\mathcal{P}_1$, $\mathcal{P}_2$, ... , $\mathcal{P}_n$, respectively, while rules of $C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ select among these all fixpoints that are simultaneously fixpoints of $\mathcal{P}_1$, $\mathcal{P}_2$, ... , $\mathcal{P}_n$.

Definition 4. *Given a set of positive programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$ over the same set of atomic propositions Var, $C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is the program over $Var' = \bigcup_{1 \leq i \leq n} \{a_{\mathcal{P}_i} \mid a \in Var\} \cup \{fail\}$ defined as follows:*

$$C(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \{fail \leftarrow not\ fail, a_{\mathcal{P}_i}, not\ a_{\mathcal{P}_j} \mid 1 \leq i \neq j \leq n\}.$$

Moreover, the program $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ over $\bigcup_{1 \leq i \leq n} Var(\Gamma(\mathcal{P}_i)) \cup \{fail\}$ is defined as:

$$J(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \Gamma(\mathcal{P}_1) \cup \cdots \cup \Gamma(\mathcal{P}_n) \cup C(\mathcal{P}_1, \cdots, \mathcal{P}_n).$$

The next theorem states that there is a one-to-one correspondence between the set of joint fixpoints of the programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$ and the set of stable models of the program $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$.

Theorem 1. *Let $\mathcal{P}_1, \cdots, \mathcal{P}_n$ be positive logic programs over the same set of atomic propositions Var. Then:*

$$SM(J(\mathcal{P}_1, \cdots, \mathcal{P}_n)) = \bigcup_{F \in JFP(\mathcal{P}_1, \ldots, \mathcal{P}_n)} \{\cup_{1 \leq i \leq n} [F]_{\mathcal{P}_i}\},$$

where $JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is the set of the joint fixpoints of $\mathcal{P}_1, \cdots, \mathcal{P}_n$.

2.5 Complexity Results

It was shown in [4] that it is NP complete to determine whether a single non-positive logic program has a fixpoint. The next table briefly introduces some decision problems and describe their complexity. The reader may find further details in [1].

Problem	Instance	Question	Complexity
JFP (JFP existence)	A set of positive logic programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$ defined over the same set of propositional variables.	Is $JFP(\mathcal{P}_1, \cdots \mathcal{P}_n) \neq \emptyset$, i.e., do the programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$ have a joint fixpoint?	NP complete
MJFP^s (skeptical reasoning under the JFP semantics)	A set of positive logic programs $S = \{\mathcal{P}_1, ..., \mathcal{P}_n\}$ defined over the same set of propositional variables Var and an atom $p \in Var$.	Does it hold that $p \in I$? For all the minimal joint fixpoints I such that $I \in MJFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$	co-NP complete
MJFP^c (credulous reasoning under the JFP semantics)	A set of positive logic programs $S = \{\mathcal{P}_1, ..., \mathcal{P}_n\}$ defined over the same set Var of propositional variables and an atom $p \in Var$.	Does it hold that $p \in I$? For some minimal joint fixpoint I such that $I \in MJFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$	Σ_2^P-complete

3 Framework Implementation

In this section we describe in detail the implementation of the framework.
Our tool has a main front-end accepting in input a list of COLP programs and computing the joint fixpoints of such programs. This result is accomplished through several stages as depicted in Figure 1.

In particular the overall process can be segmented into three big steps:

1. Converting a given set of COLP programs into classic propositional logic programs.
2. Generating a single logic program whose stable models are in one-to-one correspondence with the joint fixpoints of the input COLP programs.

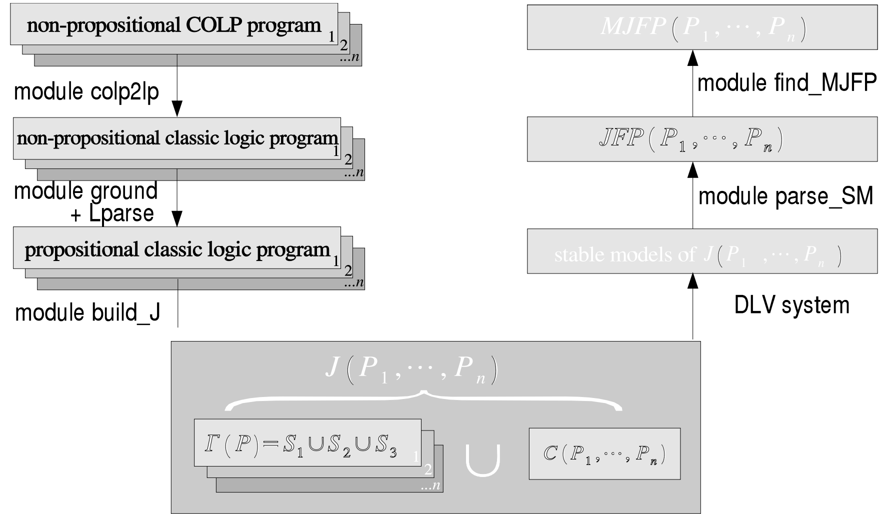


Fig. 1. JFP → SM mapping stages and software modules implemented

3. Exploiting the DLV system in order to compute the stable models of such a program and then computing the (minimal) joint fixpoints.

The first step is accomplished by the following two modules:

3.1 The Module *colp2lp*

Input: a list of non propositional COLP programs.
Output: a list of non propositional classic logic programs.

This module translates a non propositional COLP program into a classic logic program, exploiting the conversion rules described in [1]. In particular all *okay* and *okay_group* predicates are recognized and accordingly expanded into classic rules. Moreover each integrity constraint is mapped to a rule whose head atom doesn't occur in any other program. With regard to the beginning example, the COLP P_{Dad} is translated into the following classic logic program:

```
twin_airbag ←
 abs_system ←
       city ← city
  high_disp ← high_disp, diesel
   air_cond ← air_cond, high_disp
      metal ← metal
         cd ← cd
```

As a further example, the statement *okay_group*(metal,cd) included in the program $P_{Charlie}$ is converted into the following couple of rules:

```
metal ← metal, cd
   cd ← metal, cd
```

3.2 The Module *ground*

Input: a list of non propositional classic logic programs.
Output: a list of propositional classic logic programs.

The mapping proposed in [1] works only with propositional programs. As a consequence, in order to give practical relevance to the implementation, we have to deal with the non trivial issue of grounding. Observe that if we have in input a set of non propositional COLP programs then the straight application of the mapping presented in Section 2.4 may produce *unsafe* rules. A program rule is *safe* if each variable occurring in that rule appears in at least one positive literal in the body of the same rule [7, 8]: in particular the rules included in the sets S_1, S_3 and C are possibly unsafe because variables may occur as negative literals inside the body of generated rules. Thus, for each input non propositional program it is necessary to produce the corresponding propositional version. By exploiting the tool *Lparse* [9–13] this module performs the required grounding on a given set of non propositional programs. However, a further issue has to be considered: *Lparse* is optimized to discard those rules whose instantiations have unsatisfiable bodies, i.e. when some literal in the body is not deducible. This kind of optimization is not applicable in case of the JFP semantics, where *okay* and *okay_group* predicates generate rules discarded by *Lparse* but meaningful w.r.t. the JFP semantics. Let us show the above issue by the example of chat forum presented in [1]. In this example we have a chat forum involving a fixed set of users: *Ann*, *Bob*, *Connie* and *Dan*. Each user can specify complex requirements concerning the presence of other users in the forum. The following COLP program represents the requirements of the user *Ann*:

```
                                  in_forum(ann) ←
                             okay(in_forum(dan)) ←
okay_group(in_forum(bob),in_forum(connie)) ← subject(soccer)
```

This program models the following specifications: Ann wants to enter in the forum and she tolerates the presence of Dan, but she does not require him. Moreover, the joint presence of Bob and Connie is tolerated, but only if soccer is a subject of the forum.

The above *okay* and *okay_group* predicates are translated as follows:

```
   in_forum(dan) ← in_forum(dan)
   in_forum(bob) ← in_forum(bob), in_forum(connie), subject(soccer)
in_forum(connie) ← in_forum(bob), in_forum(connie), subject(soccer)
```

Finally, the knowledge about each user is enriched by a common knowledge base, defining the relationships `user`, `day` and `subject`. Furthermore the constraint that a chat forum must contain at least two users is also included:

```
           user(ann) ←
           user(bob) ←
        user(connie) ←
           user(dan) ←
     subject(soccer) ←
         day(monday) ←
                   ⊥ ← not multiple_chat
      multiple_chat ← in_forum(X), in_forum(Y), user(X), user(Y), X ≠ Y
```

A straight use of *Lparse* in order to perform the grounding does not return a complete instantiation for the above non propositional rule because not all the in_forum predicates are deducible from the program.

This problem has been solved by enriching each non propositional program with a set of facts, each fact representing a literal occurring in the whole collection of logic programs. This way, *Lparse* is forced to generate a full instantiation for each non propositional rule. Finally we discard from each grounded program those facts which were previously absent in the non propositional version. Thus we obtain a propositional logic program with a complete instantiation (here we only show the relevant part):

```
multiple_chat ← in_forum(ann), in_forum(bob), user(ann), user(bob)
multiple_chat ← in_forum(ann), in_forum(connie), user(ann), user(connie)
multiple_chat ← in_forum(ann), in_forum(dan), user(ann), user(dan)
multiple_chat ← in_forum(bob), in_forum(ann), user(bob), user(ann)
multiple_chat ← in_forum(bob), in_forum(connie), user(bob), user(connie)
multiple_chat ← in_forum(bob), in_forum(dan), user(bob), user(dan)
multiple_chat ← in_forum(connie), in_forum(ann), user(connie), user(ann)
multiple_chat ← in_forum(connie), in_forum(bob), user(connie), user(bob)
multiple_chat ← in_forum(connie), in_forum(dan), user(connie), user(dan)
multiple_chat ← in_forum(dan), in_forum(ann), user(dan), user(ann)
multiple_chat ← in_forum(dan), in_forum(bob), user(dan), user(bob)
multiple_chat ← in_forum(dan), in_forum(connie), user(dan), user(connie)
```

3.3 The Module *build_J*

Input: a collection of propositional logic programs.
Output: a single logic program $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ whose stable models are in one-to-one correspondence with the joint fixpoints of the input COLP programs.

This modules implements the translation rules described in Section 2.4. Briefly, for each program $\mathcal{P}_i \in \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}$ the sets of rules S_1, S_2 and S_3 are generated. Afterwards the set $C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is created. Finally:

$$J(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \bigcup_{\mathcal{P}_i \in \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}} \left(S_1(\mathcal{P}_i) \cup S_2(\mathcal{P}_i) \cup S_3(\mathcal{P}_i) \right) \bigcup C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$$

is produced as a final result. In the following paragraphs the translation algorithm is shown and commented.

The Translation Algorithm

Algorithm *Translate*
Input Var: Set of atoms, $P = \{\mathcal{P}_1, \cdots, \mathcal{P}_i, \cdots, \mathcal{P}_n\}$: Set of logic programs over Var;
Output $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$: a program associated to the set P such that the stable models of $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ correspond to the joint fixpoints of $\mathcal{P}_1, \cdots, \mathcal{P}_n$;
var $S_1, S_2, S_3, C, \Gamma(\mathcal{P}_1), \cdots, \Gamma(\mathcal{P}_i), \cdots, \Gamma(\mathcal{P}_n)$: Set of rules, $LSet$, L: List of atoms;
begin
1. **for each** integer $i \mid \exists$ a program $\mathcal{P}_i \in P$ **do**
(* S_1 and S_3 are created *)
2. $S_1 = \emptyset$; $S_2 = \emptyset$; $S_3 = \emptyset$;
3. **for each** atom $a \in Var$ **do**
4. **let** a_1 be a new literal; $a_1.setName(a.getName + \text{“}p\text{”} + int2string(i))$;
5. **let** a_2 be a new literal; $a_2.setName(a.getName + \text{“ }'\text{ ”} + \text{“}p\text{”} + int2string(i))$;
6. **let** r be a new rule; $r.setHead([a_1.getName])$; $r.setBody([not\ a_2.getName])$;
7. **let** s be a new rule; $s.setHead([a_2.getName])$; $s.setBody([not\ a_1.getName])$;
8. $S_1 = S_1 \cup \{r\} \cup \{s\}$;
9. **let** a_3 be a new literal; $a_3.setName(\text{“}s\text{”} + a.getName + \text{“}p\text{”} + int2string(i))$;
10. **let** a_4 be a new literal; $a_4.setName(\text{“}fail\text{”} + \text{“}p\text{”} + int2string(i))$;
11. **let** t be a new rule; $t.setHead([a_4.getName])$;
12. $t.setBody([not\ a_4.getName, a_3.getName, not\ a_1.getName])$;
13. **let** u be a new rule; $u.setHead([a_4.getName])$;
14. $u.setBody([not\ a_4.getName, a_1.getName, not\ a_3.getName])$;
15. $S_3 = S_3 \cup \{t\} \cup \{u\}$;
16. **end for**
(* S_2 is created*)
17. **for each** rule $r \in \mathcal{P}_i \mid r : a \leftarrow b_1, \cdots, b_n$ **do**
18. **let** l be a new literal; $l.setName(\text{“}s\text{”} + a.getName + \text{“}p\text{”} + int2string(i))$;
19. $LSet = \emptyset$;
20. **for each** $b_j \in Body(r)$
21. **let** l_j be a new literal; $l_j.setName(b_j.getName + \text{“}p\text{”} + int2string(i))$;
22. **let** $LSet = LSet \cup \{l_j\}$;
23. **end for**
24. **let** $L = SetToList(LSet)$;
25. **let** s be a new rule; $s.setHead([l.getName])$; $s.setBody(L)$;
26. $S_2 = S_2 \cup \{s\}$;
27. **end for**
(* $\Gamma(\mathcal{P}_i)$ is created*)
28. $\Gamma(\mathcal{P}_i) = S_1 \cup S_2 \cup S_3$;
29. **end for**
(* $C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is created *)
30. $C = \emptyset$;
31. **let** f be a new literal; $f.setName(\text{“}fail\text{”})$;
32. **for each** atom $a \in Var$ **do**
33. **for each** integer $i \in [1, n]$ **do**
34. **let** $a_{\mathcal{P}_i}$ be a new literal; $a_{\mathcal{P}_i}.setName(a.getName + \text{“}p\text{”} + int2string(i))$;
35. **for each** integer $j \in [1, n]$ **do**
36. **if** $i \neq j$ **then**
37. **let** $a_{\mathcal{P}_j}$ be a new literal; $a_{\mathcal{P}_j}.setName(a.getName + \text{“}p\text{”} + int2string(j))$;
38. **let** r be a new rule; $r.setHead([f.getName])$;
39. $r.setBody([not\ f.getName, a_{\mathcal{P}_i}.getName, not\ a_{\mathcal{P}_j}.getName])$;
40. $C = C \cup \{r\}$;
41. **end if**
42. **end for**
43. **end for**
44. **end for**
(* $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is created *)
45. $J = \emptyset$;
46. **for each** integer $i \in [1, n]$ **do**
47. $J = J \cup \Gamma(\mathcal{P}_i)$;
48. **end for**
49. $J = J \cup C$;
end.

Comments to the Algorithm

W.l.o.g. we assume the logic programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$ being defined over the same set Var. Furthermore we assume Var being part of the input. A more general scenario has been implemented where each logic program can be defined on a different set of atoms $Var(\mathcal{P}_i)$ not being part of the input, but built at run-time. Moreover the set Var is generated as the union of all the sets $Var(\mathcal{P}_i)$.

lines 2 - 8: For each program $\mathcal{P}_i$ these statements build the set S_1:

lines 2 - 5: For each atom $a \in Var$ two literals $a_{\mathcal{P}_i}$ and $a'_{\mathcal{P}_i}$ are created. Atoms and literals are treated as objects having an attribute *name*. We assume that two methods $getName$ and $setName$ exist which are used to read / write such attribute. The function $int2string$ returns the string representation of an integer number.

lines 6 - 8: For each atom $a \in Var$ two rules $a_{\mathcal{P}_i} \leftarrow not\ a'_{\mathcal{P}_i}$ and $a'_{\mathcal{P}_i} \leftarrow not\ a_{\mathcal{P}_i}$ are created. Rules are treated as objects having two attributes: a *Head* and a *Body*. The method *setHead* (resp. *setBody*) receives a list $L = [l_1, \cdots, l_n]$ of literals and sets the head (resp. the body) of a specified rule using the literals inside L. After being created the rules are added to the set S_1.

lines 9 - 16: For each program $\mathcal{P}_i$ these statements build the set S_3:

lines 9 - 10: For each atom $a \in Var$ two literals $sa_{\mathcal{P}_i}$ and $fail_{\mathcal{P}_i}$ are created.

lines 11 - 16: For each atom $a \in Var$ two rules $fail_{\mathcal{P}_i} \leftarrow not\ fail_{\mathcal{P}_i}, sa_{\mathcal{P}_i}, not\ a_{\mathcal{P}_i}$ and $fail_{\mathcal{P}_i} \leftarrow not\ fail_{\mathcal{P}_i}, a_{\mathcal{P}_i}, not\ sa_{\mathcal{P}_i}$ are created. Finally those rules are added to the set S_3.

lines 17 - 27: For each program $\mathcal{P}_i$, the set S_2 is created. In particular:

lines 18 - 23: Each rule r of the form: $a \leftarrow b_1, \cdots, b_n$ is renamed to a new rule $s:\ sa_{\mathcal{P}_i} \leftarrow b^1_{\mathcal{P}_i}, \cdots, b^n_{\mathcal{P}_i}$. A set *LSet* is used to temporarily store the literals from the body of s.

lines 24 - 27: The set *LSet* is converted to a list L which is input to the method *setBody*. Then the rule s is added to the set S_2.

line 28: For each program $\mathcal{P}_i$ a new program $\Gamma(\mathcal{P}_i) = S_1 \cup S_2 \cup S_3$ is created.

lines 30 - 44: The set $C = C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is created. In particular for each atom $a \in Var$ and for each couple of different programs $\mathcal{P}_i$ and $\mathcal{P}_j$ two new literals $a_{\mathcal{P}_i}, a_{\mathcal{P}_j}$ and a rule of the form $fail \leftarrow not\ fail, a_{\mathcal{P}_i}, not\ a_{\mathcal{P}_j}$ are created. Then those rules are added to C.

lines 45 - 49: $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is created as $J = \Gamma(\mathcal{P}_1) \cup \cdots \cup \Gamma(\mathcal{P}_n) \cup C$.

At this stage we exploit the DLV system [7, 8] in order to compute the stable models of $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$.

3.4 The Module *parse_SM*

Input: the stable models of the program $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$.
Output: the joint fixpoints of $\mathcal{P}_1, \cdots, \mathcal{P}_n$.

This module post-processes the output results from DLV. In particular it filters the stable models of $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ extracting the relevant atoms, i.e. those being part of JFP$(\mathcal{P}_1, \cdots, \mathcal{P}_n)$. Finally the original names those atoms had within the COLP programs are restored, i.e. name prefixes and suffixes added by the translation process are discarded. For example, a stable model of $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is the following:

{s_abs_system_P1, s_twin_airbag_P1, s_twin_airbag_P2, abs_system_P1, air_cond_1_P1, cd_P1, city_P1, diesel_1_P1, high_disp_1_P1, metal_P1, petrol_P1, twin_airbag_P1, s_cd_P1, s_city_P1, s_metal_P1, s_petrol_P1, abs_system_P2, air_cond_1_P2, auto_shift_1_P2, cd_P2, city_P2, diesel_1_P2, high_disp_1_P2, metal_P2, petrol_P2, steering_1_P2, twin_airbag_P2, s_abs_system_P2, s_cd_P2, s_city_P2, s_metal_P2, s_petrol_P2, abs_system_P3, air_cond_1_P3, auto_shift_1_P3, cd_P3, city_P3, diesel_1_P3, high_disp_1_P3, metal_P3, petrol_P3, steering_1_P3, twin_airbag_P3, s_abs_system_P3, s_cd_P3, s_city_P3, s_metal_P3, s_petrol_P3, s_twin_airbag_P3}

where all atoms having a prefix s_ (the supported atoms) jointly occur with those without the prefix (the fixpoints) as an effect of the rules in S_3. Moreover the suffix _Pi indicates which logic program an atom comes from.

From this stable model the following joint fixpoint is extracted:
{abs_system, twin_airbag, cd, city, metal, petrol}

3.5 The Module *find_MJFP*

Input: the joint fixpoints of $\mathcal{P}_1, \cdots, \mathcal{P}_n$.
Output: the minimal joint fixpoints of $\mathcal{P}_1, \cdots, \mathcal{P}_n$.

The set of minimal joint fixpoints MJFP$(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is built in a bottom-up fashion. At the beginning MJFP is empty and the input joint fixpoints are sorted by ascending cardinality, i.e. the number of included atoms. Fixpoints having the minimum cardinality are also minimal, thus they are directly included in MJFP. The remaining fixpoints are separately processed: a fixpoint is discarded if an element of MJFP exists which is included in it, otherwise it is added to MJFP. In the following paragraphs the algorithm is shown and commented.

3.6 MJFP Search Algorithm

Algorithm *MJFP Search*
Input $JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$: Set of joint fixpoints of the programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$.
Output $MJFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$: Set of minimal joint fixpoints of the programs $\mathcal{P}_1, \cdots, \mathcal{P}_n$.
var L: List of fixpoints; *mincard*: Integer; *discard*: Boolean;
begin
1. **for each** element $f \in JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ **do**
2. $L.append(f)$;
3. **end for**
4. *sort*(L, cardinality, ascending);
5. *mincard* = *card*($L.first$); $MJFP = \emptyset$;
6. **for each** element l in list L **do**
7. **if** *card*(l) = *mincard* **then** $MJFP = MJFP \cup \{l\}$;
8. **else begin**

```
 9.                 discard = false;
10.                 for each element m ∈ MJFP do
11.                     if {m} ⊆ {l} then discard = true;
12.                 end for
13.                 if not discard then MJFP = MJFP ∪ {l};
14.             end;
15.     end for
end.
```

Comments to the Algorithm

lines 1 - 3: The elements of $JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ are copied in a list L.
line 4: The elements of L are sorted by ascending cardinalities, i.e. number of included atoms.
lines 6 - 15: This is the core of the algorithm, in particular:
- **line 7:** Fixpoints having the minimum cardinality are added to MJFP.
- **lines 8 - 12:** Fixpoints not having the minimum cardinality are checked for set inclusion minimality: if any of them includes an element of MJFP then it is discarded.
- **lines 13 - 15:** Fixpoints which are not discarded are minimal, so they are added to MJFP.

Furthermore, even though the algorithm works in polynomial time, space complexity represents a concrete drawback. In fact all joint fixpoints have to be computed before finding the minimal ones, i.e. exponential space is required. As discussed in the next section, an interesting issue to be investigated is of course the problem of encoding minimality directly into the original programs in order to avoid space exploitation generated by the previous approach. However, this approach may have significance every time the number of JFPs is small, that is a plausible case, since it corresponds to have heterogeneous agents' requirements.

4 Conclusions and Future Work

In this paper we described a framework which implements the joint fixpoint semantics introduced in [1] on the top of the DLV system. This semantics is suitable to model joint decisions of agents represented by logic programs. In order to meet this target we realized software tools which receive in input a collection of COLP programs, each representing the requirements and desires of an agent, and generate a single program whose stable models are in one-to-one correspondence to the COLPs' joint fixpoints. Those stable models are computed exploiting the DLV system. As a final result we compute the joint fixpoints of the input COLP programs, i.e. the fixpoints which are common to all the input COLP programs. Those joint fixpoints represent a common agreement among the agents. Moreover we embedded an option to compute the minimal joint fixpoints, i.e. the joint fixpoints containing the minimum number of atoms required to meet a common agreement between the agents. It is easy to see that the computational complexity of the implementation reaches the theoretic one. It has

been pointed out that searching for the minimal joint fixpoints requires an exponential space, thus a new scheme for the mapping from joint fixpoint semantics to stable model semantics has to be investigated in order to directly produce the minimal joint fixpoints. Grounding is another issue that could be solved in an alternative way: again, by changing the translation scheme it could be possible to avoid the generation of unsafe rules within the sets S_1, S_3 and C. This way the translation process could directly work with non propositional COLP programs, thus avoiding the overhead produced by the ground instantiations. This is left for future investigations.

References

1. Buccafurri, F., Gottlob, G.: Multiagent Compromises, Joint Fixpoints and Stable Models. *Computational Logic: From Logic Programming into the Future (In honour of Bob Kowalsky)*, Ed. A. Kakas and F. Sadri, Springer Verlag (2002)
2. Baral C., Gelfond M.: Logic Programming and Knowledge Representation. *Journal of Logic Programming* **19-20**, (1994) 73–148
3. Minker J., Seipel D.: Disjunctive Logic Programming: A Survey and Assessment. *Computational Logic: From Logic Programming into the Future (In honour of Bob Kowalsky)*, Ed. A. Kakas and F. Sadri, Springer Verlag (2002)
4. Kolaitis, P.G., Papadimitriou, C.H.: Why not Negation by Fixpoint? *Journal of Computer and System Sciences* **43**(1), (1991) 125–144
5. Gelfond, M., Lifschitz, V.: The Stable Model Semantics for Logic Programming. *Proceedings of the 5th International Conference on Logic Programming*, MIT Press, Cambridge, (1988) 1070–1080
6. Baral C.: Knowledge Representation, Reasoning and Declarative Problem Solving. Cambridge University Press, (2003)
7. Leone, N., Pfeifer, G., Faber, W., Calimeri, F., Dell'Armi, T., Eiter, T, Gottlob, G., Ianni, G., Ielpa, G., Koch, C., Perri, S., and Polleres, A.: The DLV System. *Proceedings of the 8th European Conference on Logics in Artificial Intelligence (JELIA) n.2424 in Lecture Notes in Computer Science*, (2002) 537–540
8. Bihlmeyer, R., Faber, W., Ielpa, G., and Pfeifer, G.: DLV - User Manual. Available at: http://www.dlvsystem.com
9. Syrjänen, T.: Lparse 1.0 User's Manual. Available at: http://www.tcs.hut.fi/Software/smodels/lparse.ps.gz
10. Niemelä, I., Simons, P.: Smodels - an Implementation of the Stable Model and Well-founded Semantics for Normal Logic Programs. *Proceedings of the 4th International Conference on Logic Programming and Non-Monotonic Reasoning (LPNMR '97), Lecture Notes in Computer Science, Vol. 1265*. Springer-Verlag, Dagstuhl, Germany (1997) 420–429
11. Niemelä, I., Simons, P., and Syrjänen, T.: Smodels: a system for answer set programming. *Proceedings of the 8th International Workshop on Non-Monotonic Reasoning*, Breckenridge, Colorado, USA, (2000)
12. Syrjänen, T.: Implementation of Local Grounding for Logic Programs with Stable Model Semantics. Technical Report, Helsinki University of Technology, Digital Systems Laboratory, (1998)
13. Simons, P.,Niemelä, I.,and Soinen, T.: Extending and Implementing the stable model semantics. *Artificial Intelligence 138*, (2002)

A compositional Semantics for CHR

Maurizio Gabbrielli[1] and Maria Chiara Meo[2]

[1] Dipartimento di Scienze dell'Informazione, Università di Bologna,
Mura A.Zamboni 7, 40127 Bologna, Italy
gabbri@cs.unibo.it
[2] Dipartimento di Scienze, Università di Chieti
Viale Pindaro 42, 65127 Pescara, Italy
cmeo@unich.it

Abstract.
Constraint Handling Rules (CHR) is a committed-choice declarative language which has been designed for writing constraint solvers. A CHR program consists of multi-headed guarded rules which allow to rewrite constraints into simpler ones until a solved form is reached.
CHR has received a considerable attention, both from the practical and from the theoretical side. Nevertheless, due the use of multi-headed clauses, there are several aspects of the CHR semantics which are not been clarified yet. In particular, no compositional semantics for CHR has been defined so far.
In this paper we introduce a fix-point semantics which characterizes the input/output behavior of a CHR program and which is and-compositional, that is, which allows to retrieve the semantics of a conjunctive query from the semantics of its components. Such a semantics can be used as a basis to define incremental and modular analysis and verification tools.

1 Introduction

Constraint Handling Rules (CHR) [9, 10] is a committed-choice declarative language which has been specifically designed for writing constraint solvers. The first constraint logic languages used mainly built-in constraint solvers designed by following a "black box" approach. This made hard to modify, debug, and analyze a specific solver. Moreover, it was very difficult to adapt an existing solver to the needs of some specific applications, and this was soon recognized as a serious limitation since often practical applications involve application specific constraints.

By using CHR one can easily introduce specific user-defined constraints and the related solver into an host language. In fact, a CHR program consists of (a set of) multi-headed guarded simplification and propagation rules which are specifically designed to implement the two most important operations involved in the constraint solving process: Simplification rules allow to replace constraints by simpler ones, while preserving their meaning. Propagation rules are used to add new redundant constraints which do not modify the meaning of the given constraint and which can be useful for further reductions. It is worth noting that the presence of multiple heads in CHR is an essential feature which is needed in order to define reasonably expressive constraint solvers (see the discussion in [10]). However, such a feature, which differentiates this proposal from

many existing committed choice logic languages, complicates considerably the semantics of CHR, in particular it makes very difficult to obtain a compositional semantics, as we argue below. This is unfortunate, as compositionality is an highly desirable property for a semantics. In fact, a compositional semantics provides the basis to define incremental and modular tools for software analysis and verification, and these features are essential in order to deal with partially defined components. Moreover, in some cases, modularity allows to reduce the complexity of verification of large systems by considering separately smaller components.

In this paper we introduce a fix-point semantics for CHR which characterizes the input/output behavior of a program and which is and-compositional, that is, which allows to retrieve the semantics of a conjunctive query from the semantics of its components.

In general, due to the presence of synchronization mechanisms, the input/ouput semantics is not compositional for committed choice logic languages and for most concurrent languages in general. Indeed, the need of more complicate semantic structures based on traces was recognized very early as a necessary condition to obtain a compositional model, first for dataflow languages [11] and then in the case of many other paradigms, including imperative concurrent languages [7] and concurrent constraint and logic languages [5].

When considering CHR this basic problem is further complicated: due to the presence of multiple heads the traces consisting of sequences of input/ouput pairs, analogous to those used in the above mentioned works, are not sufficient to obtain a compositional semantics. Intuitively the problem can be stated as follows. A CHR rule $r:\ A,B \Leftrightarrow c \mid C$ cannot be used to rewrite a goal A, no matter how the variables are constrained (that is, for any input constraint), because the goal consists of a single atom A while the head of the rule contains two atoms A,B. Therefore, if we considered a semantics based on input/ouput traces, we would obtain the empty denotation for the goal A in the program consisting of the rule r plus some rules defining C. Analogously for the goal B. On the other hand, the rule r can be used to rewrite the goal A,B. Therefore, provided that the semantics of C is not empty, the semantics of A,B is not empty and cannot be derived from the semantics of A and B, that is, such a semantics is not compositional. It is worth noting that even restricting to a more simple notion of observable, such as the results of terminating computations, does not simplify this problem. In fact, differently from the case of ccp languages, also the semantic based on these observables (usually called resting points) is not compositional for CHR.

Our solution to obtain a compositional model is to use an augmented semantics based on traces which includes at each steps two "assumptions" on the external environment and two "outputs" of the current process: Similarly to the case of the models for ccp, the first assumption is made on the constraints appearing in the guards of the rules, in order to ensure that these are satisfied and the computation can proceed. The second assumption is specific to our approach and contains atoms which can appear in the heads of rules. This allows us to rewrite a goal G by using a rule whose head H properly contains G: While this is not possible with the standard CHR semantics, we allow that by assuming that the external environment provides the "difference" H minus G and by memorizing such an assumption. The first output element is the constraint produced by the process, as usual. We also memorize at each step also a second output

element, consisting of those atoms which are not rewritten in the current derivation and which could be used to satisfy some assumptions (of the second type) when composing sequences representing different computations. Thus our model is based on sequences of quadruples, rather than of simple input/output pairs.

Our compositional semantics is obtained by a fixpoint construction which uses an enhanced transitions system implementing the rules for assumptions described above. We prove the correctness of the semantics w.r.t. a notion of observables which characterizes the input/ouput behavior of terminating computations where the original goal has been completely reduced to built-in constraints. We will discuss later the extensions needed in order to characterize different notions of results, such as the "qualified answers" used in [10].

The remaining of this paper is organized as follows. Next section introduces some preliminaries about CHR and its operational semantics. Section 3 contains the definition of the compositional semantics, while section 4 presents the compositionality and correctness results. Section 5 concludes by discussing directions for future work.

2 Preliminaries

In this section we first introduce some preliminary notions and then define the CHR syntax and operational semantics. Even though we try to provide a self-contained exposition, some familiarity with constraint logic languages and first order logic could be useful.

We first need to distinguish the constraints handled by an existing solver, called built-in (or predefined) constraints, from those defined by the CHR program, called CHR (or user defined) constraints. An atomic constraint is a first-order predicate (atomic formula). By assuming to use two disjoint sorts of predicate symbols we then distinguish built-in atomic constraints from CHR atomic constraints. A built-in constraint c is defined by

$$c ::= a | c \wedge c | \exists_x a$$

where a is an atomic built-in constraint[1]. For built-in constraints we assume given a theory CT which defines their meaning.

On the other hand, according to the usual CHR syntax, we assume that a CHR constraint is a conjunction of atomic CHR constraints. We use c, d to denote built-in constraints, g, h, k to denote CHR constraints and a, b to denote both built-in and CHR constraints (we will call these generically constraints). The capital versions of these notations will be used to denote multisets of constraints. Furthermore we denote by $\mathcal{U}$ the set of user defined constraints and by $\mathcal{B}$ the set of built-in constraints.

We will often use "," rather than $\wedge$ to denote conjunction and we will often consider a conjunction of atomic constraints as a multiset of atomic constraints. In particular, we will use this notation based on multisets in the syntax of CHR. The notation $\exists_{-V}\phi$ where V is a set of variables denotes the existential closure of a formula ϕ with the

[1] We could consider more generally first order formulas as built-in constraints, as far as the results presented here are concerned.

exception of the variables V which remain unquantified. $Fv(\phi)$ denotes the free variables appearing in ϕ and we denote by $\cdot$ the concatenation of sequences and by ε the empty sequence. Furthermore $\uplus$ denotes the multi-set union, while we consider $\setminus$ as an overloaded operator used both for set and multi-set difference (the meaning depends on the type of the arguments).

We are now ready to define the CHR syntax.

Definition 1 (Syntax). *[10] A CHR simplification rule has the form $H \Leftrightarrow c \mid B$ while a CHR propagation rule has the form $H \Rightarrow c \mid B$ where H is a non-empty multiset of user-defined constraints, c is a built-in constraint and B is a possibly empty multi-set of constraints.*

A CHR program is a set of CHR simplification and propagation rules. A CHR goal is a multiset of (both user defined an built-in) constraints.

We prefer to use multisets rather than sequences (as in the original CHR papers) since multisets appear to correspond more precisely to the nature of CHR rules. We denote by $Goals$ the set of all goals.

We describe now the operational semantics of CHR as provided by [10] by using a transition system $T_s = (Conf_s, \longrightarrow_s)$ ($_s$ here stands for "standard", as opposed to the semantics we will use later). Configurations in $Conf_s$ are triples of the form $\langle G, K, d\rangle$ where G are the constraints that remain to be solved, K are the user-defined constraints that have been accumulated and d are the built-in constraints that have been simplified [2].

An *initial configuration* has the form

$$\langle G, \emptyset, \emptyset\rangle$$

and consists of a query G, an empty user-defined constraint and an empty built-in constraint.

A *final configuration* has either the form

$$\langle G, K, \{\texttt{false}\}\rangle,$$

when it is *failed*, i.e. when it contains an inconsistent built-in constraint store represented by the unsatisfiable constraint `false`, or has the form

$$\langle \emptyset, K, d\rangle$$

when it is successfully terminated, i.e. there are no goals left to solve.

Given a program P, the transition relation $\longrightarrow_s \subseteq Conf \times Conf$ is the least relation satisfying the rules in Table 1 (for the sake of simplicity, we omit indexing the relation with the name of the program). The **Solve** transition allows to update the constraint store by taking into account a built-in constraint contained in the goal. Without loss of generality, we will assume that $Fv(d') \subseteq Fv(c) \cup Fv(d)$. The **Introduce** transition is

[2] In [10] triples of the form $\langle G, K, d\rangle_{\mathcal{V}}$ were used, where the annotation $\mathcal{V}$, which is not changed by the transition rules, is used to distinguish the variables appearing in the initial goal from the variables which are introduced by the rules. We can avoid such an indexing by explicitly referring to the original goal.

Solve	$\dfrac{CT \models c \wedge d \leftrightarrow d' \text{ and c is a built-in constraint}}{\langle (c, G), K, d\rangle \longrightarrow_s \langle G, K, d'\rangle}$
Introduce	$\dfrac{\text{h is a user-defined constraint}}{\langle (h, G), K, d\rangle \longrightarrow_s \langle G, (h, K), d\rangle}$
Simplify	$\dfrac{H \Leftrightarrow c \mid B \in P \quad x = Var(H) \quad CT \models d \rightarrow \exists_x((H = H') \wedge c)}{\langle G, H' \wedge K, d\rangle \longrightarrow_s \langle B \wedge G, K, H = H' \wedge d\rangle}$
Propagate	$\dfrac{H \Rightarrow c \mid B \in P \quad x = Var(H) \quad CT \models d \rightarrow \exists_x((H = H' \wedge c)}{\langle G, H' \wedge K, d\rangle \longrightarrow_s \langle B \wedge G, H' \wedge K, H = H' \wedge d\rangle}$

Table 1. The standard transition system for CHR

used to move a CHR constraint from the goal to the CHR constraint store, where it can be handled by applying CHR rules. The transitions **Simplify** and **Propagate** allow to rewrite CHR constraints (which are in the CHR constraint store) by using rules from the program. As usual, in order to avoid variable names clashes, both these transitions assume that clauses from the program are renamed apart, that is assume that all variables appearing in a program clause are fresh ones. Both the **Simplify** and **Propagate** transitions are applicable when the current store (d) is strong enough to entail the guard of the rule (c), once the parameter passing has been performed (this is expressed by the equation $H = H'$). Note that, due to the existential quantification over the variables x appearing in H, in such a parameter passing the information flow is from the actual parameter (in H') to the formal parameters (H), that is, it is required that the constraints H' which have to be rewritten are an instance of the head H. When applied, both these transitions add the body B of the rule to the current goal and the equation $H = H'$, expressing the parameter passing mechanism, to the built-in constraint store. The difference between **Simplify** and **Propagate** is in the fact that while the former transition removes the constraints H' which have been rewritten from the CHR constraint store, this is not the case for the latter.

Given a goal G, the operational semantics that we consider observes the final stores of computations terminating with an empty goal and an empty user-defined constraint. We call these observables success answers slightly deviating from the terminology of [10] (a goal which has a success answer is called a data-sufficient goal in [10]).

Definition 2 (Success answers). *Let P be a program and let G be a goal. The set $\mathcal{SA}_P(G)$ of success answers for the query G in the program P is defined as follows*

$$\mathcal{SA}_P(G) = \{\langle \exists_{-Fv(G)} d\rangle \mid \langle G, \emptyset, \emptyset\rangle \longrightarrow_s^* \langle \emptyset, \emptyset, d\rangle \not\longrightarrow_P\}.$$

In [10] it is considered also the following different notion of answer, obtained by computations terminating with a user-defined constraint which does not need to be empty.

Definition 3 (Qualified answers). *Let P be a program and let G be a goal. The set $\mathcal{QA}_P(G)$ of qualified answers for the query G in the program P is defined as follows*

$$\mathcal{QA}_P(G) = \{\langle \exists_{-Fv(G)} K \wedge d\rangle \mid \langle G, \emptyset, \emptyset\rangle \longrightarrow_s^* \langle \emptyset, K, d\rangle \not\longrightarrow_P\}.$$

We discuss in Section 5 the extensions needed to characterize also qualified answers. Note that both previous notions of observables characterize an input/output behavior, since the input constraint is implicitly considered in the goal.

In the remaining of this paper we will consider only simplification rules since propagation rules can be mimicked by simplification rules, as far as the results contained in this paper are concerned.

3 A compositional trace semantics

Given a program P, we say that a semantics $\mathcal{S}_P$ is and-compositional if $\mathcal{S}_P(A, B) = \mathcal{C}(\mathcal{S}_P(A), \mathcal{S}_P(B))$ for a suitable composition operator $\mathcal{C}$ which does not depend on the program P. As mentioned in the introduction, due to the presence of multiple heads in CHR, the semantics which associate to a program P the function $\mathcal{SA}_P$ is not and-compositional, since goals which have the same input/ouput behavior can behave differently when composed with other goals. Consider for example the program P consisting of the single rule

$$g, h \Leftrightarrow true|c.$$

(where c is a built-in constraint). According to Definition 3 we have that $\mathcal{SA}_P(g) = \mathcal{SA}_P(k) = \emptyset$, while $\mathcal{SA}_P(g, h) = \{\langle \exists_{-Fv(g,h)} c\rangle\} \neq \emptyset = \mathcal{SA}_P(k, h)$. An analogous example can be made to show that also the semantics $\mathcal{QA}$ is not and-compositional.

The problem exemplified above is different from the classic problem of concurrent languages where the interaction of non-determinism and synchronization makes the input/outuput observables non-compositional. For this reason, considering simply sequences of (input-output) built-in constraints is not sufficient to obtain a compositional semantics for CHR. We have to use some additional information which allow us to describe the behavior of goals in any possible and-composition without, of course, considering explicitly all the possible and-compositions.

The basic idea of our approach is to collect in the semantics also the "missing" parts of heads which are needed in order to proceed with the computation. For example, when considering the program P above, we should be able to state that the goal g produces the constraint c, provided that the external environment (i.e. a conjunctive goal) contains the CHR constraint h. In other words, h is an assumption which is made in the semantics describing the computation of g. When composing (by using a suitable notion of composition) such a semantics with that one of a goal which contains h we can verify that the "assumption" h is satisfied and therefore obtain the correct semantics for g, h. In order to model correctly the interaction of different processes we have to use sequences, analogously to what happens with other concurrent paradigms.

This idea is developed in the following by defining a new transition system which implements this mechanism based on assumptions for dealing with the missing parts of heads. The new transition system allows to generate the sequences appearing in the

Solve'	$\dfrac{CT \models c \wedge d \leftrightarrow d'}{\langle c \wedge G^{max=i}, d\rangle \longrightarrow^{\emptyset} \langle G^{max=i}, d'\rangle}$
Simplify'	$\dfrac{\mathsf{cl} = H \leftrightarrow c \mid B \in P \quad x = Fv(H) \quad G^{max=i} \neq \emptyset \quad CT \models d \rightarrow \exists_x((H = (G^{max=i}, K)) \wedge c)}{\langle G^{max=i} \wedge A, d\rangle \ \longrightarrow^{K} \langle B^{i+1} \wedge A, d \wedge (H = (G, K))\rangle}$

Table 2. The transition system for the compositional semantics

compositional model by using a standard fix-point construction. As a first step in our construction we modify the notion of configuration used before: Since we do not need to distinguish user defined constraints which appear in the goal from the user defined constraints which have been already considered for reduction, we merge the first and the second components of previous triples. On the other hand, we need the information on the new assumptions, which is added as a label of the transitions.

Thus we define a transition system $T = (\mathit{Conf}, \longrightarrow_P)$ where configurations in Conf are pairs: the first component is a multi-set of indexed atoms (the goal) and the second one is a built-in constraint (the store). Indexes are associated to atoms in order to denote the point in the derivation where they have been introduced. More precisely, atoms in the original goals are labeled by 0, while atoms introduced at the $i - th$ derivation step are labeled by i. Given a program P, the transition relation $\longrightarrow_P \subseteq \mathit{Conf} \times \mathit{Conf} \times \wp(\mathcal{U})$ is the least relation satisfying the rules in Table 2. Note that we consider only **Solve** and **Simplify** rules, as the other rules as previously mentioned are redundant in this context. **Solve'** is the same rule as before, while the **Simplify'** rule is modified to consider assumptions: When reducing a goal G by using a rule having head H, the set of assumption $K = H \setminus G$ (with $H \neq K$) is used to label the transition ($\setminus$ here denotes multiset difference). Indexes allow us to distinguish identical occurrences of atoms which have been introduced in different derivation steps. We will use the notation $G^{max=i}$ to indicate that i is the maximal label occurring in the (non-atomic) goal G and G^i to indicate that all the atoms in G are labeled by i. When no ambiguity arise, to simplify the notation we will use G^i also to denote $G^{max=i}$. In particular, the notation $\langle G^i, c\rangle \rightarrow \langle B^{i+1}, d\rangle$ used in some cases in the transitions always means $\langle G^{max=i}, c\rangle \rightarrow \langle B^{i+1}, d\rangle$, that is, if i is the maximal label occurring in G then all the atoms in B are labeled by $i + 1$. When indexes are not needed we will simply omit them. As before, we assume that program rules to be used in the new simplify rule use fresh variables to avoid names captures.

The semantics domain of our compositional semantics is based on sequences which represent derivations obtained by the transition system in Table 2. More precisely, we first consider "concrete" sequences consisting of tuples of the form $\langle G, c, K, G', d\rangle$: Such a tuple represents exactly a derivation step $\langle G, c\rangle \longrightarrow^{K} \langle G', d\rangle$. The sequences we consider are terminated by tuples of the form $\langle G, c, \emptyset, G, c\rangle$, which represent a terminating step (see the precise definition below). Since a sequence represents a derivation, we assume that the "output" goal G' at step i is equal to the "input" goal G at step $i+1$,

that is, we assume that if

$$\ldots\langle G_i, c_i, K_i, G'_i, d_i\rangle\langle G_{i+1}, c_{i+1}, K_{i+1}, G'_{i+1}, d_{i+1}\rangle\ldots$$

appears in a sequence, then $G'_i = G_{i+1}$ holds.

On the other hand, the input store c_{i+1} can be different from the output store d_i produced at previous step, since we need to perform all the possible assumptions on the constraint c_{i+1} produced by the external environment in order to obtain a compositional semantics. However, we assume that if

$$\ldots\langle G_i, c_i, K_i, G'_i, d_i\rangle\langle G_{i+1}, c_{i+1}, K_{i+1}, G'_{i+1}, d_{i+1}\rangle\ldots$$

appears in a sequence then $CT \models c_{i+1} \rightarrow d_i$ holds: This means that the assumption made on the external environment cannot be weaker than the constraint store produced at the previous step. This reflects the monotonic nature of computations, where information can be added to the constraint store and cannot be deleted from it. Finally note that assumptions on CHR constraints (label K) are made only for the atoms which are needed to "complete" the current goal in order to apply a clause. In other words, no assumption can be made in order to apply clauses whose heads do not share any predicate with the current goal.

The set of the above described "concrete" sequences, which represent derivation steps performed by using the new transition system, is denoted by $\mathcal{S}eq$.

From these concrete sequences we extract some more abstract sequences which are the objects of our semantic domain: From each tuple $\langle G, c, K, G', d\rangle$ in a sequence $\delta \in \mathcal{S}eq$ we extract a tuple of the form $\langle c, K, H, d\rangle$ where we consider as before the input and output store (c and d, respectively) and the assumptions (K), while we do not consider anymore the output goal G'. Furthermore, we restrict the input goal G to that part H consisting of all those user-defined constraints which will not be rewritten in the (derivation represented by the) sequence δ. Intuitively H contains those atoms which are available for satisfying assumptions of other goals, when composing two different sequences (representing two derivations of different goals). We also assume that if $\langle c_i, K_i, H_i, d_i\rangle\langle c_{i+1}, K_{i+1}, H_{i+1}, d_{i+1}\rangle$ is in a sequence then $H_i \subseteq H_{i+1}$ holds, since these atoms which will not be rewritten in the derivation can only augment. We then define formally the semantic domain as follows.

Definition 4 (Sequences). *The semantic domain $\mathcal{D}$ containing all the possible sequences is defined as the set*

$$\begin{aligned}\mathcal{D} = \{\langle c_1, K_1, H_1, d_1\rangle \ldots \langle c_n, \emptyset, H_n, c_n\rangle \mid \\ &\textit{for each } j,\ 1 \leq j \leq n \textit{ and for each } i, 1 \leq i \leq n-1, \\ &H_j \textit{ and } K_i \textit{ are multisets of CHR indexed constraints,} \\ &c_j, d_i \textit{ are built-in constraints such that } CT \models d_i \rightarrow c_i, \\ &H_i \subseteq H_{i+1} \textit{ and } CT \models c_{i+1} \rightarrow d_i\}.\end{aligned}$$

In order to define our semantics we need two more notions. First, we define an abstraction operator α which extracts from the concrete sequences in $\mathcal{S}eq$ (representing exactly derivation steps) the sequences used in our semantic domain.

Definition 5. *Let $\delta = \langle G_1, c_1, K_1, G_2, d_1\rangle \dots \langle G_n, c_n, \emptyset, G_n, c_n\rangle$ be a sequence of derivation steps where we assume that atoms are indexed as previously specified. We say that an indexed atom A^j is stable in δ if A^j appears in G_i and in G_{i+1}, for each $1 \leq i \leq n-1$. The abstraction operator $\alpha : \mathcal{S}eq \to \mathcal{D}$ is then defined inductively as*

$$\begin{aligned} &\alpha(\varepsilon) = \varepsilon \\ &\alpha(\langle G, c, K, G', d\rangle \cdot \delta') = \langle c, K, H, d\rangle \cdot \alpha(\delta') \end{aligned}$$

where H is the multiset consisting of all the indexed atoms in G which are stable in $\langle G, c, K, G', d\rangle \cdot \delta'$.

Then we need the notion of "compatibility" of a tuple w.r.t. a sequence. To this aim we first provide some further notation: Given a sequence of derivation steps

$$\delta = \langle G_1, c_1, K_1, G_2, d_1\rangle\langle G_2, c_2, K_2, G_3, d_2\rangle \dots \langle G_n, c_n, \emptyset, G_n, c_n\rangle$$

we denote by $length(\delta)$ and by $instore(\delta)$ the length of the derivation δ and the first input store c_1, respectively. Moreover using t as a shorthand for the tuple $\langle G_1, c_1, K_1, G_2, d_1\rangle$ we define

$V_{loc}(t) = Fv(G_2, d_1) \setminus Fv(G_1, c_1, K_1)$,
$V_{ing}(\delta) = Fv(G_1)$,
$V_{ass}(\delta) = \bigcup_{i=1}^{n-1} Fv(K_i)$,
$V_{stable}(\delta) = Fv(G_n)$,
$V_{constr}(\delta) = \bigcup_{i=1}^{n-1} Fv(d_i) \setminus Fv(c_i)$ and
$V_{loc}(\delta) = \bigcup_{i=1}^{n-1} Fv(G_{i+1}, d_i) \setminus Fv(G_i, c_i, K_i)$.

We then define a compatibility as follows.

Definition 6. *Let $t = \langle G_1, c_1, K_1, G_2, d_1\rangle$ a tuple representing a derivation step for the goal G_1 and let $\delta = \langle G_2, c_2, K_2, G_3, d_2\rangle \dots \langle G_n, c_n, \emptyset, G_n, c_n\rangle$ be a sequence of derivation steps for G_2. We say that t is compatible with δ if the following hold:*

1. *$CT \models instore(\delta) \to d_1$,*
2. *$V_{loc}(\delta) \cap Fv(t) = \emptyset$,*
3. *for $i \in [2, n]$, $V_{loc}(t) \cap Fv(c_i) \subseteq \bigcup_{j=1}^{i-1} Fv(d_j) \cup V_{stable}(\delta)$ and*
4. *$V_{loc}(t) \cap V_{ass}(\delta) = \emptyset$.*

Note that if t is compatible with δ then, by using the notation above, $t \cdot \delta$ is a sequence of derivation steps for G_1. We can now define the compositional semantics.

Definition 7 (Compositional semantics). *Let P be a program and let G be a goal. The compositional semantics of G in the program P, $\mathcal{S}_P : Goals \to \wp(\mathcal{D})$, is defined as*

$$\mathcal{S}_P(G) = \alpha(\mathcal{S}'_P(G))$$

where α is the operator introduced in Definition 5 and $\mathcal{S}'_P : Goals \to \wp(\mathcal{S}eq)$ is defined as follows:

$$\begin{aligned}\mathcal{S}'_P(G) = \{\langle G, c, K, G', d\rangle \cdot \delta \in \mathcal{S}eq \mid\ & CT \not\models c \leftrightarrow \mathtt{false},\ \langle G, c\rangle \longrightarrow^K \langle G', d\rangle \\ & \textit{and } \delta \in \mathcal{S}_P(G') \textit{ for some } \delta \textit{ such that} \\ & \langle G, c, K, G', d\rangle \textit{ is compatible with } \delta\} \\ \cup\ & \\ \{\langle G, c, \emptyset, G, c\rangle \in \mathcal{S}eq\}.&\end{aligned}$$

Formally $\mathcal{S}'_P(G)$ is defined as the least fixed-point of the corresponding operator $\Phi \in (Goals \to \wp(\mathcal{S}eq)) \to Goals \to \wp(\mathcal{S}eq)$ defined by

$$\begin{aligned}\Phi(I)(G) = \{\langle G, c, K, G', d\rangle \cdot \delta \in \mathcal{S}eq \mid\ & CT \not\models c \leftrightarrow \mathtt{false},\ \langle G, c\rangle \longrightarrow^K \langle G', d\rangle \\ & \textit{and } \delta \in I(G') \textit{ for some } \delta \textit{ such that} \\ & \langle G, c, K, G', d\rangle \textit{ is compatible with } \delta\} \\ \cup\ & \\ \{\langle G, c, \emptyset, G, c\rangle \in \mathcal{S}eq\}.&\end{aligned}$$

In the above definition, the ordering on $Goals \to \wp(\mathcal{S}eq)$ is that of (point-wise extended) set-inclusion. It is straightforward to check that Φ is continuous (on a CPO), thus standard results ensure that the fixpoint can be calculated by $\sqcup_{n\geq 0}\phi^n(\bot)$, where ϕ^0 is the identity map and for $n > 0$, $\phi^n = \phi \circ \phi^{n-1}$ (see for example [8]).

4 Compositionality and correctness

In this section we prove that the semantics defined above is and-compositional and correct w.r.t. the observables $\mathcal{SA}_P$.

In order to prove the compositionality result we first need to define how two sequences describing a computation of A and B, respectively, can be composed in order to obtain a computation of A, B. Such a composition is defined by the (semantic) operator $||$ which performs an interleaving of the actions described by the two sequences and then eliminates the assumptions which are satisfied in the resulting sequence. For technical reasons, rather than modifying the existing sequences, the elimination of satisfied assumptions is performed on new sequences which are generated by a closure operator η defined as follows.

Definition 8. *Let W be a multiset of indexed atoms and let σ be a sequence in $\mathcal{D}$ of the form*

$$\langle c_1, K_1, H_1, d_1\rangle \dots \langle c_n, K_n, H_n, d_n\rangle.$$

We denote by $\sigma \setminus W$ the sequence

$$\langle c_1, K_1, H_1 \setminus W, d_1\rangle \dots \langle c_n, K_n, H_n \setminus W, d_n\rangle$$

where the multisets difference $H_i \setminus W$ considers indexes.

The operator $\eta : \wp(\mathcal{D}) \to \wp(\mathcal{D})$ is defined as follows. Given $S \in \wp(\mathcal{D})$, $\eta(S)$ is the least set satisfying the following conditions:

1. $S \subseteq \eta(S)$;
2. if $\sigma' \cdot \langle c, K, H, d\rangle \cdot \sigma'' \in S$ *then* $(\sigma' \cdot \langle c, K \setminus K', H, d\rangle \cdot \sigma'') \setminus W \in \eta(S)$

where $K' = \{A_1, \ldots, A_n\} \subseteq K$ *is a multiset such that there exists a multiset (of indexed atoms)* $W = \{B_1^{j_1}, \ldots, B_n^{j_n}\} \subseteq H$ *such that* $CT \models c \wedge B_i \leftrightarrow c \wedge A_i$*, for each* $i \in [1, n]$.

A few explanations are in order. The operator η is an upper closure operator [3] which saturates a set of sequences S by adding new sequences where redundant assumptions can be removed: an assumptions a (in K_i) can be removed if a^j appears as a stable atom (in H_i). Once a stable atom is "consumed" for satisfying an assumption it is removed from (the sets of stable atoms of) all the tuples appearing in the sequence, to avoid multiple uses of the same atom. Note that stable atoms are considered without the index in the condition $CT \models c \wedge B_i \leftrightarrow c \wedge A_i$, while they are considered as indexed atoms in the removal operation $H_i \setminus W$. The reason for this slight complication is explained by the following example. Assume that we have the set S consisting of the only sequence $\langle c, \emptyset, \{a^1\}, d\rangle\langle c', \{a\}, \{a^1, a^2\}, d'\rangle$. Such a sequence indicates that at the second step we have an assumption a, while both at the first and at the second step we have produced a stable atom a, which has been indexed by 1 and 2, respectively. In order to satisfy the assumption a we can use either a^1 or a^2. However, depending on what indexed atom we use, we obtain two different simplified sequences in $\eta(S)$, namely $\langle c, \emptyset, \emptyset, d\rangle\langle c', \emptyset, \{a^2\}, d'\rangle$ and $\langle c, \emptyset, \{a^1\}, d\rangle\langle c', \emptyset, \{a^1\}, d'\rangle$, which describes correctly the two different situations.

Before defining the composition operator $\|$ on sequences we need a notation for the sequences in $\mathcal{D}$ analogous to that one introduced for sequences of derivation steps:
Let $\sigma = \langle c_1, K_1, H_1, d_1\rangle\langle c_2, K_2, H_2, d_2\rangle \cdots \langle c_n, \emptyset, H_n, d_n\rangle \in \mathcal{D}$ be a sequence for the goal G. We define

$V_{ing}(\sigma) = Fv(G)$ (the free variables of the goal G),
$V_{ass}(\sigma) = \bigcup_{i=1}^{n-1} Fv(K_i)$ (the variables in the assumptions of σ),
$V_{stable}(\sigma) = Fv(H_n) = \bigcup_{i=1}^{n} Fv(H_i)$ (the variables in the stable multisets of σ),
$V_{constr}(\sigma) = \bigcup_{i=1}^{n-1} Fv(d_i) \setminus Fv(c_i)$ (the variables in the output constraints of σ which are not in the corresponding input constraints),
$V_{loc}(\sigma) = (V_{constr}(\sigma) \cup V_{stable}(\sigma)) \setminus (V_{ass}(\sigma) \cup V_{ing}(\sigma))$.

We can now define the composition operator $\|$ on sequences. To simplify the notation we denote by $\|$ both the operator acting on sequences and that one acting on sets of sequences.

Definition 9. *The operator* $\| : \mathcal{D} \times \mathcal{D} \to \wp(\mathcal{D})$ *is defined inductively as follows. Assume that* $\sigma_1 = \langle c_1, K_1, H_1, d_1\rangle \cdot \sigma_1'$ *and* $\sigma_2 = \langle c_2, K_2, H_2, d_2\rangle \cdot \sigma_2'$ *are sequences for the goals* G_1 *and* G_2*, respectively. If*

$$(V_{loc}(\sigma_1) \cup Fv(G_1)) \cap (V_{loc}(\sigma_2) \cup Fv(G_2)) = Fv(G_1) \cap Fv(G_2)$$

then $\sigma_1 \| \sigma_2$ *is defined by cases as follows:*

[3] $S \subseteq \eta(S)$ holds by definition, and it is easy to see that $\eta(\eta(S)) = \eta(S)$ holds and that $S \subseteq S'$ implies $\eta(S) \subseteq \eta(S')$.

1. *If both σ_1 and σ_2 have length* 1*, say* $\sigma_1 = \langle c, \emptyset, H_1, c\rangle$ *and* $\sigma_2 = \langle c, \emptyset, H_2, c\rangle$*, then*

$$\sigma_1 \parallel \sigma_2 = \{\langle c, \emptyset, H_1 \uplus H_2, c\rangle\}.$$

2. *If σ_2 has length* 1 *and σ_1 has length* > 1 *then*

$$\sigma_1 \parallel \sigma_2 = \{\langle c_1, K_1, H_1 \uplus H_2, d_1\rangle \cdot \sigma \mid \sigma \in \sigma_1' \parallel \sigma_2 \text{ and } CT \models instore(\sigma) \rightarrow d_1\}.$$

The symmetric case is analogous and therefore omitted.

3. *If both σ_1 and σ_2 have length* > 1 *then*

$$\begin{aligned}\sigma_1 \parallel \sigma_2 = \{&\langle c_1, K_1, H_1 \uplus H_2, d_1\rangle \cdot \sigma \in \mathcal{D} \mid \sigma \in (\sigma_1' \parallel \sigma_2) \\ &\qquad\qquad\qquad \text{and } CT \models instore(\sigma) \rightarrow d_1\} \\ \cup& \\ \{&\langle c_2, K_2, H_1 \uplus H_2, d_2\rangle \cdot \sigma \in \mathcal{D} \mid \sigma \in (\sigma_1 \parallel \sigma_2') \\ &\qquad\qquad\qquad \text{and } CT \models instore(\sigma) \rightarrow d_2\}\end{aligned}$$

Finally the composition of sets of sequences $\parallel$: $\wp(\mathcal{D}) \times \wp(\mathcal{D}) \rightarrow \wp(\mathcal{D})$ *is defined by:*

$$\begin{aligned}S_1 \parallel S_2 = \{\sigma \in \mathcal{D} \mid\ & \text{there exist } \sigma_1 \in S_1 \text{ and } \sigma_2 \in S_2 \text{ such that} \\ & \sigma = \langle c_1, K_1, H_1, d_1\rangle \cdots \langle c_n, \emptyset, H_n, c_n\rangle \in \eta(\sigma_1 \parallel \sigma_2), \\ & (V_{loc}(\sigma_1) \cup V_{loc}(\sigma_2)) \cap V_{ass}(\sigma) = \emptyset \text{ and for } i \in [1, n] \\ & (V_{loc}(\sigma_1) \cup V_{loc}(\sigma_2)) \cap Fv(c_i) \subseteq \textstyle\bigcup_{j=1}^{i-1} Fv(d_j) \cup \bigcup_{j=1}^{i} Fv(H_j)\}.\end{aligned}$$

Using this notion of composition of sequences we can show that the semantics $\mathcal{S}_P$ is compositional. Before proving the compositionality theorem we need some technical lemmas.

Lemma 1. *Let G be a goal, $\delta \in \mathcal{S}_P'(G)$ and let $\sigma = \alpha(\delta)$. Then $V_r(\delta) = V_r(\sigma)$ holds, where $r \in \{\, ing,\ ass,\ stable,\ constr,\ loc\,\}$.*

Lemma 2. *Let P be a program, G_1 and G_2 be two goals and assume that $\delta \in \mathcal{S}_P'(G_1, G_2)$. Then there exists $\delta_1 \in \mathcal{S}_P'(G_1)$ and $\delta_2 \in \mathcal{S}_P'(G_2)$, such that $\alpha(\delta) \in \eta(\alpha(\delta_1) \parallel \alpha(\delta_2))$.*

Lemma 3. *Let P be a program, let G_1 and G_2 be two goals and assume that $\delta_1 \in \mathcal{S}_P'(G_1)$ and $\delta_2 \in \mathcal{S}_P'(G_2)$ are two sequences such that the following hold:*

1. *$\alpha(\delta_1) \parallel \alpha(\delta_2)$ is defined,*
2. *$\sigma = \langle c_1, K_1, W_1, d_1\rangle \cdots \langle c_n, \emptyset, W_n, c_n\rangle \in \eta(\alpha(\delta_1) \parallel \alpha(\delta_2))$,*
3. *$(V_{loc}(\alpha(\delta_1)) \cup V_{loc}(\alpha(\delta_2))) \cap V_{ass}(\sigma) = \emptyset$,*
4. *for $i \in [1, n]$, $(V_{loc}(\alpha(\delta_1)) \cup V_{loc}(\alpha(\delta_2))) \cap Fv(c_i) \subseteq \bigcup_{j=1}^{i-1} Fv(d_j) \cup \bigcup_{j=1}^{i} Fv(W_j)$.*

Then there exists $\delta \in \mathcal{S}_P'(G_1, G_2)$ such that $V_{loc}(\delta) \subseteq V_{loc}(\delta_1) \cup V_{loc}(\delta_2)$ and $\sigma = \alpha(\delta)$.

By using the above results we can prove the following theorem.

Theorem 1 (Compositionality). *Let P be a program and let G_1 and G_2 be two goals. Then*

$$\mathcal{S}_P(G_1, G_2) = \mathcal{S}_P(G_1) \parallel \mathcal{S}_P(G_2).$$

4.1 Correctness

In order to show the correctness of the semantics $\mathcal{S}_P$ w.r.t. the (input/output) observables $\mathcal{SA}_P$, we first introduce a different characterization of $\mathcal{SA}_P$ obtained by using the new transition system defined in Table 2.

Definition 10. *Let P be a program and let G be a goal and let $\longrightarrow$ be (the least relation) defined by the rules in Table 2. We define*

$$\mathcal{SA}'_P(G) = \{\exists_{-Fv(G)}c \mid \langle G, \emptyset\rangle \longrightarrow^*_P \langle \emptyset, c\rangle \not\longrightarrow_P\}.$$

The correspondence of $\mathcal{SA}'$ with the original notion $\mathcal{SA}$ is stated by the following proposition, whose proof is immediate.

Proposition 1. *Let P be a program and let G be a goal. Then*

$$\mathcal{SA}_P(G) = \mathcal{SA}'_P(G).$$

The observables $\mathcal{SA}'_P$, and therefore $\mathcal{SA}_P$, describing answers of successful computations can be obtained from $\mathcal{S}$ by considering suitable sequences, namely those sequences which do not perform assumptions neither on CHR constraints nor on built-in constraints. The first condition means that the second components of tuples must be empty, while the second one means that the assumed constraint at step i must be equal to the produced constraint of steps i-1. We call "connected" those sequences which satisfy these requirements:

Definition 11 (Connected sequences). *Assume that*

$$\sigma = \langle c_1, K_1, H_1, d_1\rangle \dots \langle c_n, K_n, H_n, c_n\rangle$$

is a sequence in $\mathcal{D}$. We say that σ is connected if $K_j = \emptyset$ for each j, $1 \le j \le n$ and $d_i = c_{i+1}$, for each i, $1 \le i \le n-1$.

The proof of the following result derives from the definition of connected sequence and an easy inductive argument. Given a sequence $\sigma = \langle c_1, K_1, H_1, d_1\rangle \dots \langle c_n, K_n, H_n, d_n\rangle$, we denote by $store(\sigma)$ the built-in constraint d_n, by $result(\sigma)$ the constraint $d_n \wedge H_n$ and by $lastg(\sigma)$ the goal H_n.

Proposition 2. *Let P be a program and let G be a goal. Then*

$$\begin{aligned}\mathcal{SA}'_P(G) = \{\exists_{-Fv(G)}c \mid\ & \textit{there exists } \sigma \in \mathcal{S}_P(G) \textit{ such that } instore(\sigma) = \emptyset,\\ & \sigma \textit{ is connected, } lastg(\sigma) = \emptyset \textit{ and } c = store(\sigma)\}.\end{aligned}$$

The following corollary is immediate from Proposition 1.

Corollary 1 (Correctness). *Let P be a program and let G be a goal. Then*

$$\begin{aligned}\mathcal{SA}_P(G) = \{\exists_{-Fv(G)}c \mid\ & \textit{there exists } \sigma \in \mathcal{S}_P(G) \textit{ such that } instore(\sigma) = \emptyset,\\ & \sigma \textit{ is connected, } lastg(\sigma) = \emptyset \textit{ and } c = store(\sigma)\}.\end{aligned}$$

5 Conclusions

In this paper we have introduced a semantics for CHR which is compositional w.r.t. the and-composition of goals and which is correct w.r.t "success answers", a notion of observable which considers the results of successful computations where all the CHR constraints have been rewritten into built-in constraints. We are not aware of other compositional characterizations of CHR answers, so our work can be considered as a first step which can be extended along several different lines.

Firstly, it would be desirable to obtain a compositional characterization also for "qualified answers" obtained by considering computations terminating with a user-defined constraint which does not need to be empty (see Definition 3). This could be done by a slight extension of our model: The problem here is that, given a tuple $\langle G, c, K, G', d \rangle$, in order to reconstruct correctly the qualified answers we need to know whether the configuration $\langle G', d \rangle$ is terminating or not (that is, if $\langle G', d \rangle \not\rightarrow$ holds). This could be solved by introducing some termination modes, at the price of a further complication of the traces used in our semantics.

A second possible extension is the investigation of the full abstraction issue. For obvious reasons it would be desirable to introduce in the semantics the minimum amount of information needed to obtain compositionality, while preserving correctness. In other terms, one would like to obtain a results of this kind: $\mathcal{S}_P(G) = \mathcal{S}_P(G')$ if and only if, for any H, $\mathcal{SA}_P(G, H) = \mathcal{SA}_P(G', H)$ (our Corollary 1 only ensures that the "only if" part holds). Such a full abstraction result could be difficult to achieve, however techniques similar to those used in [5, 2] for analogous results in the context of ccp could be considered

It would be interesting also to study further notions of compositionality, for example that one which considers union of program rules rather than conjunctions of goals, analogously to what has been done in [6]. However, due to the presence of synchronization, the simple model based on clauses defined in [6] cannot be used for CHR.

As mentioned in the introduction, the main interest related to a compositional semantics in the possibility to provide a basis to define compositional analysis and verification tools. In our case, it would be interesting to investigate to what extent the compositional proof systems *à la* Hoare defined in [1, 3] for (timed) ccp languages, based on resting points and trace semantics, can be adapted to the case of CHR.

Acknowledgments

We thank Michael Maher for having initially suggested the problem of compositionality for CHR semantics.

References

1. F.S. de Boer, M. Gabbrielli, E. Marchiori and C. Palamidessi. Proving Concurrent Constraint Programs Correct. *Transactions on Programming Languages and Systems (TOPLAS)*, 19(5): 685-725. ACM Press, 1997.
2. F.S. de Boer, M. Gabbrielli, and M.C. Meo. Semantics and expressive power of a timed concurrent constraint language. In G. Smolka. editor, Proc. *Third Int'l Conf. on Principles and*

Practice of Constraint Programming (CP 97), Lecture Notes in Computer Science. Springer-Verlag, 1997.

3. F.S. de Boer, M. Gabbrielli and M.C. Meo. Proving correctness of Timed Concurrent Constraint Programs. *ACM Transactions on Computational Logic*. To appear.
4. F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm for asynchronous communication. In J.C.M. Baeten and J.F. Groote, editors, *Proceedings of CONCUR'91*, vol. 527 of *LNCS*, pages 111–126. Springer-Verlag, 1991.
5. F.S. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT/CAAP*, vol. 493 of *LNCS*, pages 296–319. Springer-Verlag, 1991.
6. A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science* 122(1-2): 3–47, 1994.
7. S. Brookes. A fully abstract semantics of a shared variable parallel language. In *Proc. Eighth IEEE Symposium on Logic In Computer Science*. IEEE Computer Society Press, 1993.
8. B.A. Davey and H.A. Priestley. Introduction to Lattices and Order. Cambridge University Press, 1990.
9. T. Früwirth. Introducing simplification rules. TR ECRC-LP-63, ECRC Munich. October 1991.
10. T. Früwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 1994:19, 20:1-679.
11. B. Jonsson. A model and a proof system for asynchronous processes. In *Proc. of the 4th ACM Symp. on Principles of Distributed Computing*, pages 49–58. ACM Press, 1985.
12. V.A. Saraswat. *Concurrent Constraint Programming Languages*. PhD thesis, Carnegie-Mellon University, January 1989. Published by The MIT Press, 1991.
13. V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of POPL*, pages 232–245. ACM Press, 1990.
14. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of Concurrent Constraint Programming. In *Proc. of POPL*. ACM Press, 1991.

Demos

A demonstration of *SOCS-SI*⋆

Marco Alberti[1], Federico Chesani[2], Marco Gavanelli[1], Evelina Lamma[1], Paola Mello[2], and Paolo Torroni[2]

[1] DI, Università di Ferrara (Italy)
{malberti, mgavanlli, elamma}@ing.unife.it
[2] DEIS, Università di Bologna (Italy)
{fchesani, pmello, ptorroni} @deis.unibo.it

1 Introduction

The demonstration that we propose is meant to show *SOCS-SI* [1], a logic-based tool for verification of compliance of agent interaction to protocols based on the notion of social expectation.

The inputs to *SOCS-SI* are:

– a text file containing the *Social Integrity Constraints* (SICs in the following), used for expressing interaction protocols;
– a text file containing an (abductive) logic program (*Social Organization Knowledge Base*, SOKB in the following), used to express declarative knowledge, such as the value of time deadlines;
– a *history* of events, representing the agent behaviour, recorded from a source of events. So far, the implementation of *SOCS-SI* accepts as sources (*i*) the user prompt, (*ii*) a log file, or (*iii*) a network-based tool for the observation of the agent interaction.

By means of an abductive proof procedure, *SOCS-SI* uses SICs and SOKB to generate *expectations* about the "ideal" social behaviour of agents, i.e., compliant to interaction protocols, given a (partial) history of events. *SOCS-SI* also checks if the actual agent behaviour corresponds to such expectations. Based on that, it either outputs an answer of *fulfillment* (if the agent behaviour is compliant to the interaction protocols) or *violation* (if the agent behaviour is not compliant to interaction protocols).

The demonstration will start with a very brief presentation of the notions of protocols, social semantics and expectations, and compliance. The proof procedure will then be briefly (and informally) introduced, and the functioning of the tool, where SICs, SOKB and history files can be visualized.

The scenario presented in the demonstration will be a "first price sealed bid" auction; for this scenario, we will show an example of fulfillment and one of violation.

More examples will be shown, if time allows.

⋆ This work is partially funded by the Information Society Technologies programme of the European Commission under the IST-2001-32530 project, and by the national MIUR COFIN 2003 projects "Sviluppo e verifica di sistemi multi-agente basati sulla logica" and "La gestione e la negoziazione automatica dei diritti sulle opere dell'ingegno digitali: aspetti giuridici ed informatici".

2 The *SOCS-SI* tool

The Social Infrastructure (SI) tool is an implementation of a logic programming based framework defined within the SOCS project [2]. The main idea of the framework is to exploit abduction in order to verify the compliance of agents behavior in respect to interactions protocols. Protocols are represented by means of integrity constraints, while expectations about future (and past) agent behaviors are mapped onto abducible predicates.

In order to verify compliance, a new proof procedure, *SC*-IFF, has been defined and implemented [3]. The *SC*-IFF is mainly based on the IFF abductive proof procedure [4], with some extensions:

- The set of facts (events) grows dynamically
- It deals with CLP constraints
- It implements the concepts of fulfillment and violation

SOCS-SI is implemented as a Java-Prolog application, where the abductive proof procedure is implemented in Prolog. The Graphical User Interface and the interfaces to the sources of events instead are implemented in Java. Through the GUI it is possible to observe, for each agent, what events are related to a certain agent, as well as its expectations. Different colors highlight expectations of different type: pending, fulfilled and violated. Fig. 1 shows a screenshot of *SOCS-SI*: a protocol violation has been detected, as a consequence of a wrong behavior of an agent. The figure shows the particular case in which an agent fails to take an action which it was expected to take by some temporal deadline. As the deadline expires, *SOCS-SI* promptly detects the violation.

3 Outline of the demonstration

1. (Informal) introduction to the logic-based social framework, to the formalism expressing the interaction protocols, and to the proof procedure used for verification.
2. Demonstration scenario: first price sealed bid auction. In this auction, an auctioneer announces an auction for a single item (which the auctioneer may want to sell or, as in our example, buy) to a set of agents. By some deadline, the agents may place a bid for the item. Then, the auctioneer must decide which bid is the best, and notify by some deadline both the winner and the losers.
3. Description of the SICs used for expressing the interaction protocols. For instance, the following SIC:
   ```
   H(tell(A,B,answer(win,Item,Quote),AuctionId),TWin)--->
   EN(tell(A,B,answer(lose,Item,Quote),AuctionId),TLose)
   ∧ TLose > TWin.
   ```
   expresses that the auctioneer is not allowed to notify an agent that its bid has lost after notifying that it has won. The protocol is expressed by four such SICs.

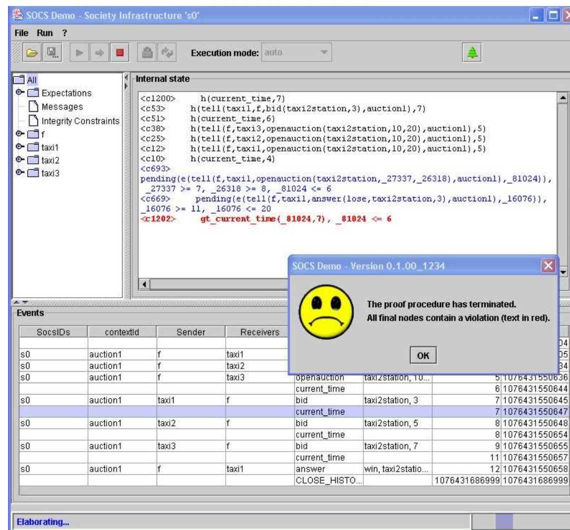


Fig. 1. The *SOCS-SI* Graphic User Interface

4. Running *SOCS-SI*: the GUI will be used to show how to select the interaction protocols and the event source, and how to observe the agent interaction and the proof procedure computation. We plan to show at least two examples: one of fulfillment (the agents will all behave as expected) and one of violation (for instance, the auctioneer may fail to notify the losers, or may notify an agent of both winning and losing the auction).

References

1. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Compliance verification of agent interaction: a logic-based tool. In Trappl, R., ed.: Proceedings of the 17th European Meeting on Cybernetics and Systems Research, Vol. II, Symposium "From Agent Theory to Agent Implementation" (AT2AI-4), Vienna, Austria, Austrian Society for Cybernetic Studies (2004) 570–575
2. Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. Home Page: `http://lia.deis.unibo.it/Research/SOCS/`.
3. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of interaction protocols: a computational logic approach based on abduction. Technical Report CS-2003-03, Dipartimento di Ingegneria di Ferrara, Ferrara, Italy (2003) `http://www.ing.unife.it/aree_ricerca/informazione/cs/technical_reports`.
4. Fung, T.H.: Abduction by Deduction. PhD thesis, Imperial College London (1996)

Two constraint-based tools for protein folding

Luca Bortolussi, Alessandro Dal Palù, and Agostino Dovier

Dip. di Matematica e Informatica, Univ. di Udine
Via delle Scienze 206, 33100 Udine (Italy).
(bortolus|dalpalu|dovier)@dimi.uniud.it

Introduction. A protein is a list of linked units called aminoacids. There are 20 different kinds of aminoacids and the typical length of a protein is less than 500 units. The Protein Structure Prediction Problem (PSP) is the problem of predicting the 3D native conformation of a protein, when its aminoacid sequence is known. The process for reaching this state is called the *protein folding*. It is widely accepted that the native conformation ensures a state of minimum free energy [1]. We assume some energy functions previously defined and we focus on the problem of finding the 3D conformation that minimizes them. We present two tools developed following different approaches to this problem. In the first we use Constraint Logic Programming over finite domains applied on the modeling of the Protein Structure Prediction problem on the Face-Centered Cubic Lattice [4]. In the second we develop a high-level off-lattice simulation method which makes use of Concurrent Constraint Programming [3]. The two codes are available at http://www.dimi.uniud.it/dovier/PF.

CLP($\mathcal{FD}$) minimization. A non-linear minimization problem can be easier to solve when the solution's space is finite. In this context, this can be done by setting admissible aminoacid's positions as the vertices of a lattice. We use the so-called *Face-Centered Cubic Lattice* [7], which is a good discrete model for protein's conformations. We look for the protein conformation in the lattice that minimizes a function which is the sum of the contributions of all pairs of aminoacids. Each contribution is non-zero only if two aminoacids are under a certain lattice distance and the precise value depends on their type as described in [2]. The tool is written in SICStus Prolog, using the `clpfd` library and it is based on [4]. We have added constraints obtained by secondary structure prediction (prediction of some local conformations, such as helices and sheets), which are currently very accurate, to reach acceptable computation time. We have also developed a local coordinate system to define torsional angles, which allows to link efficiently the secondary structure information to the three-dimensional folding. Moreover, we have implemented a method to dynamically prune the search tree based on the analysis of the contacts between the aminoacids during the folding process. The results we obtained allow us to effectively predict proteins up to 60 aminoacids. The actual version of the tool is much faster that the first version presented in [5]: for some proteins we have reached a speed up of more than 200 times. Anyway, time is still considerable, as can be expected from the NP completeness of the problem. Proteins of length up to 20 are folded correctly

in few seconds, while for longer proteins (around 40 aminoacids) the optimal solution is reached in 3 to 10 hours. Proteins of length 60 take longer time, even if acceptable solutions are found in about 10 hours (on a PC, 3 GHz, 512MB). The user can choose the maximum search time and he/she can prune the search tree imposing a "compact" coefficient that acts on the allowed 3D structure to the protein. In figure 1 it is shown the tool while working on protein 1YPA of length 63, with time limit of 24h (86400s) and compact factor of 0.17. The solution is found in 14 hours and saved on a standard "pdb" file viewed using the program ViewerLite.

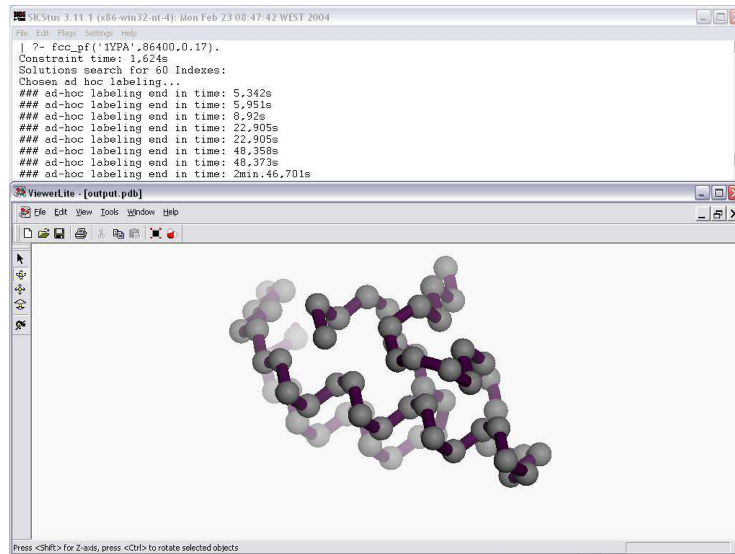


Fig. 1. CLP($\mathcal{FD}$) minimizator

CCP simulation. In this tool, described in [3], we adopt an off-lattice simplified representation of a protein, where each aminoacid is represented by a center of interaction. The empirical contact energy function [2] used in the constraint-based approach is modified and augmented by local terms which describe bond lengths, bend angles, and torsion angles. Our simulation makes use of concurrent constraint programming. Basically, each aminoacid of the protein is viewed as an independent agent that moves in the space and communicates with other aminoacids. Each agent waits for a communication of the modification of other aminoacids' position; after receiving a message, it stores the information in a list and performs a move. The new position is computed using a Montecarlo simulation, based on the spatial information available to the aminoacid, which may not be the current dislocation of the protein, due to asynchrony in the communication. Once the move is performed, the aminoacid communicates its new position to all the others. The code has been implemented in Mozart [6].

We tested our system either on known proteins or on artificial sequences of aminoacids. The code works properly on sequences of Alanines, that are known to have a high tendency to form a single helix, while for more complex structures the minima not always corresponds to the real native conformation. In Figure 2 we show the tool while folding a helix from a sequence of 14 Alanines. As initial state of the protein we set each aminoacid along a line with a step of the bond distance (3.8 Å). We run the simulation for 60 seconds on a PC, 1GHz, 256MB.

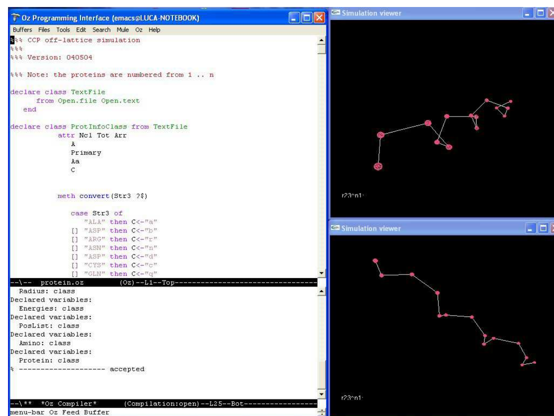


Fig. 2. CCP Simulator.

References

1. C. B. Anfinsen. Principles that govern the folding of protein chains. *Science*, 181:223–230, 1973.
2. M. Berrera, H. Molinari, and F. Fogolari. Amino acid empirical contact energy definitions for fold recognition in the space of contact maps. *BMC Bioinformatics*, 4(8), 2003.
3. L. Bortolussi, A. Dal Palù, A. Dovier, and F. Fogolari. Protein Folding Simulation in CCP. Submitted to BIOCONCUR 2004.
4. A. Dal Palù, A. Dovier, and F. Fogolari. Protein folding in $CLP(\mathcal{FD})$ with empirical contact energies. In *Recent Advances in Constraints*, volume 3010 of *Lecture Notes in Artificial Intelligence*, pp. 250–265, 2004.
5. A. Dovier, M. Burato, and F. Fogolari. Using secondary structure information for protein folding in *clp*($\mathcal{FD}$). *Proceedings of the 11th International Workshop on Functional and (Constraint) Logic Programming*, 76, 2002.
6. Univ. des Saarlandes, Sweedish Inst. of Computer Science, and Univ. Catholique de Louvain. The Mozart Programming System. www.mozart-oz.org.
7. G. Raghunathan and R. L. Jernigan. Ideal architecture of residue packing and its observation in protein structures. *Protein Science*, 6:2072–2083, 1997.

The DALI Logic Programming Agent-Oriented Language[*]

Stefania Costantini   Arianna Tocchio

Università degli Studi di L'Aquila
Dipartimento di Informatica
Via Vetoio, Loc. Coppito, I-67010 L'Aquila - Italy
{stefcost,tocchio}@di.univaq.it

1 The DALI language

DALI [3] [2] is an Active Logic Programming Language designed in the line of [6] for executable specification of logical agents. A DALI agent is a logic program that contains a particular kind of rules, reactive rules, aimed at interacting with an external environment. The reactive and proactive behavior of the DALI agent is triggered by several kinds of events: external, internal, present and past events. All the events and actions are timestamped, so as to record when they occurred. The new syntactic entities, i.e., predicates related to events and proactivity, are indicated with special postfixes (which are coped with by a pre-processor) so as to be immediately recognized while looking at a program.

The external events are syntactically indicated by the postfix *E*. When an event comes into the agent from its "external world", the agent can perceive it and decide to react. The reaction is defined by a reactive rule which has in its head that external event. The special token $:>$, used instead of $:-$, indicates that reactive rules performs forward reasoning. The agent remembers to have reacted by converting the external event into a *past event* (time-stamped). Operationally, if an incoming external event is recognized, i.e., corresponds to the head of a reactive rule, it is added into a list called EV and consumed according to the arrival order, unless priorities are specified.

The internal events define a kind of "individuality" of a DALI agent, making her proactive independently of the environment, of the user and of the other agents, and allowing her to manipulate and revise her knowledge. An internal event is syntactically indicated by the postfix *I*, and its description is composed of two rules. The first one contains the conditions (knowledge, past events, procedures, etc.) that must be true so that the reaction (in the second rule) may happen.

Internal events are automatically attempted with a default frequency customizable by means of directives in the initialization file. The user's directives can tune several parameters: at which frequency the agent must attempt the internal events; how many times an agent must react to the internal event (forever, once, twice,...) and when (forever, when triggering conditions occur, ...); how long the event must be attempted (until some time, until some terminating conditions, forever).

[*] We acknowledge support by the *Information Society Technologies programme of the European Commission, Future and Emerging Technologies* under the IST-2001-37004 WASP project.

When an agent perceives an event from the "external world", it does not necessarily react to it immediately: she has the possibility of reasoning about the event, before (or instead of) triggering a reaction. Reasoning also allows a proactive behavior. In this situation, the event is called present event and is indicated by the suffix *N*.

Actions are the agent's way of affecting her environment, possibly in reaction to an external or internal event. In DALI, actions (indicated with postfix A) may have or not preconditions: in the former case, the actions are defined by actions rules, in the latter case they are just action atoms. An action rule is just a plain rule, but in order to emphasize that it is related to an action, we have introduced the new token $:<$, thus adopting the syntax $action\ :<\ preconditions$. Similarly to external and internal events, actions are recorded as past actions.

Past events represent the agent's "memory", that makes her capable to perform future activities while having experience of previous events, and of her own previous conclusions. Past events, indicated by the suffix *P*, are kept for a certain default amount of time, that can be modified by the user through a suitable directive in the initialization file.

The DALI language has been equipped with a communication architecture consists of three levels. The first level implements a FIPA-compliant [5] communication protocol and a filter on communication, i.e. a set of rules that decide whether or not receive or send a message. The DALI communication filter is specified by means of meta-level rules defining the distinguished predicates *tell* and *told*. The second level includes a meta-reasoning layer, that tries to understand message contents, possibly based on ontologies and/or on forms of commonsense reasoning. The third level consists of the DALI interpreter.

The declarative and procedural semantics of DALI, is defined as an *evolutionary semantics*, so as to cope with the evolution of an agent corresponding to the perception of events [3]. The semantics has been generalized so as to include the communication architecture [4] by resorting to the general framework RCL (Reflective Computational Logic) [1] based on the concept of reflection principle.

Following [7] and the references therein, the operational semantics of communication is defined [4] by means of a formal dialogue game framework that focuses on the rules of dialogue, regardless the meaning the agent may place on the locutions uttered. This means, we do not want to refer to the mental states of the participants.

2 The DALI Interpreter

The DALI interpreter has been implemented in Sicstus Prolog, and includes a FIPA-compliant communication library. The DALI interpreter is in principle able to interoperate with other FIPA-compliant platforms. Presently, we have implemented interoperability with JADE, which is one of the best-known non-logical middleware for agents (namely, it is written in java). DALI agents can be distributed on the web, as the implementation of the communication primitives is based on TCP/IP.

The interpreter is composed of three main modules: (i) the *DALI active_server* module, that manages the community of DALI agents; (ii) the *DALI active_user* module,

that provides a user interface for the user to interact with the agents; (iii) the *active_dali* module, that is automatically activated by the active_server whenever an agent is created (then, there are as many copies of the active_dali module running as the existing agents).

The DALI/FIPA communication protocol consists of the main FIPA primitives, plus few new primitives which are peculiar of DALI. The code implementing the FIPA primitives is contained in the le *communication_ pa.txt* , imported by agents as a library. The DALI/FIPA communication protocol is implemented by means a piece of DALI code including suitable $tell/told$ rules. Whenever a message is received, with content part *primitive(Content,Sender)* the DALI interpreter automatically looks for a corresponding *told* rule that speci es whether the message should be accepted. Symmetrically, whenever a message should be sent, with content part *primitive(Content,Sender)* the DALI interpreter automatically looks for a corresponding *tell* rule that speci es whether the message can be actually issued. The DALI code de ni ng the DALI/FIPA protocol is contained in a separate prede ned le, *communication.txt*, imported by agents as a library. In this way, both the communication primitives and the communication protocol can be seen as “input parameters ”of the agent. Typically, a user will add new application-dependent $tell/told$ rules to the le *communication.txt*. Possibly however, both les can be replaced, thus specifying a different communication protocol.

Each DALI agent must be generated by specifying the following parameters. (a) The name of the le that contains the DALI logic program (a .txt le). (b) The name of the agent. (c) The ontology the agent adopts (a .txt le); (d) The language (e.g., Italian or English etc.) used in the communication acts; (e) The name of the le containing the communication constraints, a .txt le; as mentioned, a prede ned standard version *communication.txt* is provided. (f) The name of the communication library, a .txt le; as mentioned, a standard version *communication_ pa.txt* is provided. (g) The skills that the agent means to make explicit to the community of DALI agents (e.g., profession, hobbies, etc.). Below is an example of activation of an agent.

```
agent('demo/program/prog',gino,'demo/pippo_ontology.txt',italian,
      ['demo/communication'],['demo/communication_ pa'],[tourist]).
```

From the program le, say *prog.txt*, a pre-processing stage extracts three les. (1) The le *prog.ple*, that contains a list of the special tokens occurring in the agent program, denoting internal and external events, goals, actions, etc. (2) The le *prog.plf*, that contains a list of user-de ned directives, that the DALI environment provides for *tuning* the behavior of an agent: the user can decide for instance: the priority among events; how long to keep memory of past events, and/or upon which conditions they must be removed; the starting and terminating conditions for attempting internal events, and the frequency. (3) The le *prog.pl* which contains the code of the agent; it must be noticed that all variables are rei ed, so as to guarantee safe communication and reliable metareasoning capabilities.

3 Case studies

We are able of course to show many simple examples, aimed at illustrating the basic language features. More complex case studies can however be demonstrated.

To demonstrate the usefulness of the “internal event” and “goal” mechanisms, we have considered as a case-study the implementation of STRIPS-like planning. We can show that it is possible in DALI to design and implement this kind of planning without de ning a meta-interpreter. Rather, each feasible action is managed by the agent’s proactive behavior: the agent checks whether there is a goal requiring that action, sets up the possible subgoals, waits for the preconditions to be veri ed, performs the actions (or records the actions to be done if the plan is to be executed later), and nally arranges the postconditions.

We can generalize this example to dynamic planning, where an agent is able to recover from unwanted or unexpected situation, by suitably modifying its plan.

To explain how the lter level works, we have implemented and experimented a case-study that demonstrates how this lter is powerful enough to express sophisticated concepts such as expressing and updating the *level of trust*. Trust is a kind of social knowledge and encodes evaluations about which agents can be taken as reliable sources of information or services. We focus on a practical issues: namely, how the level of Trust in uences communication and choices of the agents.

References

1. J. Barklund, S. Costantini, P. Dell’Acqua e G. A. Lanzarone, *Re ection Principles in Computational Logic*, Journal of Logic and Computation, Vol. 10, N. 6, December 2000, Oxford University Press, UK.
2. S. Costantini. Many references about DALI and PowerPoint presentations can be found at the URLs: http://costantini.di.univaq.it/pubbls_ste .htm and http://costantini.di.univaq.it/AI2.htm.
3. S. Costantini and A. Tocchio, *A Logic Programming Language for Multi-agent Systems*, In S. Flesca, S. Greco, N. Leone, G. Ianni (eds.), *Logics in Arti cial Intelligence, Proc. of the 8th Europ. Conf., JELIA 2002*, (held in Cosenza, Italy, September 2002), LNAI 2424, Springer-Verlag, Berlin, 2002.
4. S. Costantini, A. Tocchio and A. Verticchio, *Semantic of the DALI Logic Programming Agent-Oriented Language*, submitted to *Logics in Arti cial Intelligence, Proc. of the 9th Europ. Conf., JELIA 2004*.
5. FIPA. *Communicative Act Library Speci cation* , Technical Report XC00037H, Foundation for Intelligent Physical Agents, 10 August 2001.
6. R. A. Kowalski, *How to be Arti cially Intelligent - the Logical Way*, Draft, revised February 2004, Available on line, URL http://www-lp.doc.ic.ac.uk/UserPages/staff/rak/rak.html.
7. P. Mc Burney, R. M. Van Eijk, S. Parsons, L. Amgoud, *A Dialogue Game Protocol for Agent Purchase Negotiations*, J. Autonomous Agents and Multi-Agent Systems Vol. 7 No. 3, November 2003.

The JSetL library: supporting declarative programming in Java

E. Panegai, E. Poleo, G. Rossi
Dipartimento di Matematica
Università di Parma, Parma (Italy)
panegai@cs.unipr.it, gianfranco.rossi@unipr.it

1 Introduction

In this demonstration we present a Java library, called JSetL, that offers a number of facilities to support declarative programming like those usually found in logic or functional declarative languages: logical variables, list and set data structures (possibly partially specified), unification and constraint solving over sets, nondeterminism.

Declarative programming is often associated with *constraint programming*. As a matter of fact, constraints provide a powerful tool for stating solutions as sets of equations and disequations over the selected domains, which are then solved by using domain specific knowledge, with no concern to the order in which they occur in the program.

Differently from other works (e.g., [1, 4]) in JSetL we do not restrict ourselves to constraints, but we try to provide a more comprehensive collection of facilities to support a real declarative style of programming. Furthermore, we try to keep our proposal as general as possible, to provide a general-purpose tool not devoted to any specific application.

2 Main features of JSetL

The most notable characteristics of JSetL are:

- *Logical variables.* The notion of logical variable (like that usually found in logic and functional programming languages) is implemented by the class `Lvar`. An instance of the class `Lvar` is created by the Java declaration

  ```
  Lvar VarName = new Lvar(VarNameExt,VarValue);
  ```

  where VarName is the variable name, VarNameExt is an optional external name, and VarValue is an optional value for the variable. The value of a logical variable can be of any type, provided it supports the `equals` method. Logical variables which have no value associated with are said to be *uninitialized*. A logical variable remains uninitialized until it gets a

value as the result of processing some constraint involving it (in particular, equality constraints). Constraints are the main operations through which `Lvar` objects can be manipulated. Constraints can be used to set the value of a logical variable as well as to inspect it. No constraint, however, is allowed to modify the value of a logical variable.

- *Lists* and *sets*. List and set data structures (like those in [3]) are provided by classes `Lst` and `Set`, respectively. Instances of these classes are created by declarations of the form

```
Lst LstName = new Lst(LstNameExt,LstElemValues);
Set SetName = new Set(SetNameExt,SetElemValues);
```

  where the different fields have the same meaning proposed for `Lvar` variables (obviously, transported on the new domains). `Lst` and `Set` objects are dealt with as logical variables, but limited to `Lst` and `Set` values. Moreover, list/set constructor operations are provided to build lists/sets out of their elements at run-time. The main difference between lists and sets is that, while in lists the order and repetitions of elements are important, in sets order and repetitions of elements do not matter. Both lists and sets can be *partially specified*, i.e., they can contain uninitialized logical variables in place of single elements or as a part of the list/set.

- *Unification*. JSetL provides unification between logical variables—as well as between `Lst` and `Set` objects—basically in the form of equality constraints: `O1.eq(O2)`, where `O1` and `O2` are either `Lvar`, or `Lst`, or `Set` objects, unifies `O1` and `O2`. Unification allows one both to test equality between `O1` and `O2` and to associate a value with uninitialized logical variables (lists, sets) possibly occurring in the two objects. Note that solving an equality constraint implies the ability to solve a *set unification* problem (cf., e.g., [3]).

- *Constraints*. Basic set-theoretical operations, as well as equalities, inequalities and integer comparison expressions, are dealt with as *constraints*. Constraint expressions are evaluated even if they contain uninitialized variables. Their evaluation is carried on in the context of the current collection of active constraints $\mathcal{C}$ (the *constraint store*) using a powerful (set) constraint solver, which allows us to compute with partially specified data.

  The approach adopted for constraint solving in JSetL is basically the same developed for CLP($\mathcal{SET}$) [2]. To add a constraint C to the constraint store, the `add` method of the `Solver` class can be called as follows:

```
Solver.add(C)
```

  The order in which constraints are added to the constraint store is completely immaterial. After constraints have been added to the store, one can invoke their resolution by calling the `solve` method:

```
Solver.solve()
```

The `solve` method nondeterministically searches for a solution that satisfies all constraints introduced in the constraint store. If there is no solution, a `Failure` exception is generated; otherwise the computation terminates with *success* when the first solution is found. The default action for this exception is the immediate termination of the current thread. Constraints that are not solved are left in the constraint store: they will be used later to narrow the set of possible values that can be assigned to uninitialized variables.

- *Nondeterminism.* Constraint solving in JSetL is inherently nondeterministic: the order in which constraints are added/solved does not matter (*dont care nondeterminism*); solutions for set constraints (e.g., equality, membership,...) are computed nondeterministically, using choice points and backtracking (*dont know nondeterminism*).

  A simple way to exploit nondeterminism is through the use of the `Setof` method of the `Solver` class. This method allows one to explore the whole search space of a nondeterministic computation and to collect into a set all the computed solutions for a specified logical variable `x`.

All these features strongly contribute to support a real declarative programming style in Java. In particular, the use of partially specified dynamic data structures, the ability to deal with constraint expressions disregarding the order in which they are encountered and the instantiation of the logical variables possibly occurring in them, as well as the nondeterminism "naturally" supported by operations over sets, are fundamental features to allow the language to be used as a highly declarative modeling tool.

3 Programming with JSetL

Example 3.1 *All pairs*

Check whether all elements of a set `s` *are pairs, i.e., they have the form* `[x1,x2]`*, for any* `x1` *and* `x2`*.*

In mathematical terms, a (declarative) solution for this problem can be stated as follows: $(\forall x \in s)(\exists x1, x2 \;\; x = [x1, x2])$*. The program below shows how this solution can be immediately implemented in Java using JSetL.*

```
public static void all_pairs(Set s) throws Failure {
    Lvar x1 = new Lvar();
    Lvar x2 = new Lvar();
    Lvar x = new Lvar();
    Lst pair = Lst.empty.ins1(x2).ins1(x1);

    Lvar[] LocalVars = {x1,x2};
    Solver.forall(x,s,LocalVars,x.eq(pair));

    Solver.solve();
    return;
}
```

Example 3.1 shows the declaration of three logical variables, `x1`, `x2`, `x`, and the creation of a new object of type `Lst`, representing a partially specified list of two elements. The constraints to be solved are introduced by the `forall` method, using the `LocalVars` array to specify existentially quantified (logical) variables. The call to the `solve` method allows to check satisfiability of the current collection of constraints.

The next example is a method to compute the set of all subsets of a given set. In this case we use the `add` method to introduce a new constraint and the `setof` method to get all possible solutions (not only the first one).

Example 3.2 *All solutions*

Compute the powerset of a given set `s`*.*

```
public static Set powerset(Set s) throws Failure {
    Set r = new Set();
    Solver.add(r.subset(s));
    return Solver.setof(r);
}
```

If `s` *is the set* {`'a','b'`}*, the set returned by* `powerset` *is* {{},{`'a'`},{`'b'`}, {`'a','b'`}}.

References

[1] A.Chun. Constraint programming in Java with JSolver. In *Proc. Practical Applications of Constraint Logic Programming, PACLP99*, 1999.

[2] A. Dovier, C. Piazza, E. Pontelli, and G. Rossi. Sets and constraint logic programming. *ACM TOPLAS*, 22(5), 861–931, 2000.

[3] A. Dovier, E. Pontelli, and G. Rossi. Set unification. TR-CS-001/2001, Dept. of Computer Science, New Mexico State University, USA, January 2001 (available at `www.cs.nmsu.edu/TechReports`).

[4] Neng-Fa Zhou. Building Java Applets by using DJ—a Java Based Constraint Language. Available at `www.sci.brooklyn.cuny.edu/~zhou`.

Index of Authors

Alberti M. 13, 362
Alferes J. 235
Backofen R. 103
Banti F. 235
Baldoni M. 250
Baroglio C. 250
Berardi M. 220
Blandi L. 43
Boella G. 265
Bonchi F. 202
Bortolussi L. 362
Bossi A. 310
Brogi A. 235
Buccafurri F. 330
Cadoli M. 118
Caminiti G. 330
Capotorti A. 56
Chesani F. 362
Citrigno M. 148
Cordí V. 163
Costantini S. 7, 295, 368
Cumbo C. 178
D'Antonio M. 73
Dal Palú A. 103, 362
Delzanno G. 73
Dovier A. 8, 103, 362
Endriss U. 28
Faber W. 148
Formisano A. 56, 88
Gabbrielli M. 345
Gavanelli M. 13, 362
Giannotti F. 202
Greco G. 148
Gungui I. 280
Iiritano S. 178
Lamma E. 13, 362
Leone N. 148
Majkic Z. 189
Malerba D. 220
Mancarella P. 28
Mancini T. 118
Martelli A. 250
Mascardi V. 163, 280
Mello P. 13, 362
Meo M. C. 345
Omodeo E. G. 88
Orlowska E. S. 88
Panegai E. 372
Patti V. 250
Pedreschi D. 202
Pereira L. M. 3
Pettorossi A. 325
Piazza C. 310
Pinto A. M. 3
Poleo E. 372
Policriti A. 88
Prestwich S. 133
Proietti M. 325
Rossi F. 133
Rossi G. 372
Rossi S. 310
Rullo P. 178
Sadri F. 28
Sessa M.I. 43
Terreni G. 28
Tocchio A. 295, 368
Toni F. 28
Torasso L. 217

Torre var der L. 265
Torroni P. 9, 13, 362
Varlaro A. 220
Venable B. K. 133
Verticchio A. 295
Walsh T. 133
Will S. 103

Stampato in proprio dal Dipartimento di Matematica dell'Università degli Studi di Parma in Via M. D'Azeglio 85/A, 43100 Parma, adempiuti gli obblighi di cui all'articolo 1 del D.L. 31 agosto 1945 n. 660.