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} :

```
twin_airbag ←
abs_system ←
okay(city) ←
okay(high_disp) ← diesel
okay(air_cond) ← high_disp
okay(metal) ←
okay(cd) ←
```

P_{Mom} :

```
twin_airbag ←
okay(air_cond) ←
okay(auto_shift) ←
okay(city) ←
steering ← not city
okay(abs_system) ←
okay(metal) ←
okay(cd) ←
okay(high_disp) ←
```

$P_{Charlie}$:

```
okay(air_cond) ←
okay(auto_shift) ←
okay(high_disp) ←
okay_group(metal,cd) ←
metal ← not cd
okay(city) ←
okay(steering) ←
okay(diesel) ← high_disp
okay(twin_airbag) ←
okay(abs_system) ← twin_airbag
```

Furthermore the knowledge about each user includes rules which characterize the fuel type (either petrol or diesel):

```
petrol ← not diesel
diesel ← not petrol
```

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 $okay\_group(p_1, \cdots, p_n)$ 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:

$$\begin{aligned} MJFP(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \{&F \in JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n) \mid \\ &\nexists F' \in JFP(\mathcal{P}_1, \cdots, \mathcal{P}_n) \wedge F' \subset F\}. \end{aligned}$$

$MJFP(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ 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:*

$$S_1 = \{a_{\mathcal{P}} \leftarrow not\ a'_{\mathcal{P}} \mid a \in Var(\mathcal{P})\} \cup \{a'_{\mathcal{P}} \leftarrow not\ a_{\mathcal{P}} \mid a \in Var(\mathcal{P})\}$$

$$S_2 = \{sa_{\mathcal{P}} \leftarrow b^1_{\mathcal{P}}, ..., b^n_{\mathcal{P}} \mid a \leftarrow b_1, \cdots b_n \in \mathcal{P}\}$$

$$\begin{aligned} S_3 = \{&fail_{\mathcal{P}} \leftarrow not\ fail_{\mathcal{P}}, sa_{\mathcal{P}}, not\ a_{\mathcal{P}} \mid a \in Var(\mathcal{P})\} \cup \\ &\{fail_{\mathcal{P}} \leftarrow not\ fail_{\mathcal{P}}, a_{\mathcal{P}}, not\ sa_{\mathcal{P}} \mid a \in Var(\mathcal{P})\}. \end{aligned}$$

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.

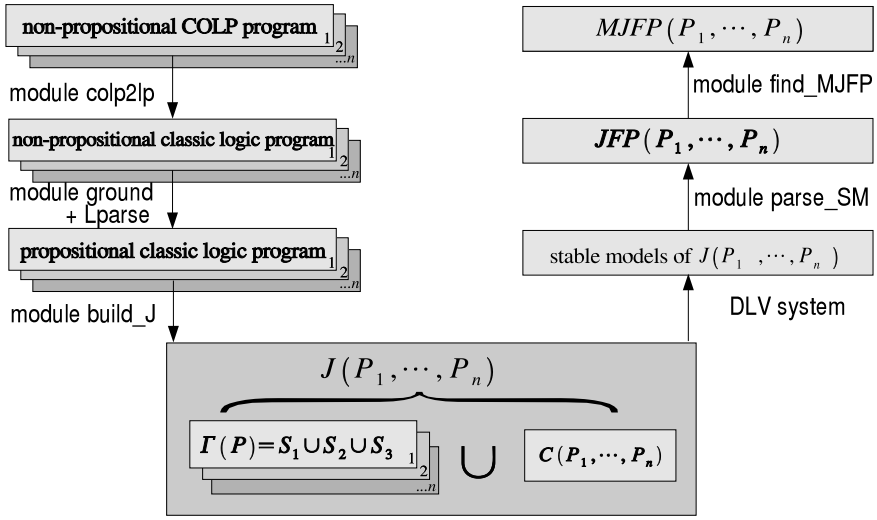

Fig. 1. JFP → SM mapping stages and software modules implemented

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*:

$$\begin{aligned} \texttt{in\_forum(ann)} &\leftarrow \\ \mathit{okay}\texttt{(in\_forum(dan))} &\leftarrow \\ \mathit{okay\_group}\texttt{(in\_forum(bob),in\_forum(connie))} &\leftarrow \texttt{subject(soccer)} \end{aligned}$$

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:

$$\begin{aligned} \texttt{in\_forum(dan)} &\leftarrow \texttt{in\_forum(dan)} \\ \texttt{in\_forum(bob)} &\leftarrow \texttt{in\_forum(bob), in\_forum(connie), subject(soccer)} \\ \texttt{in\_forum(connie)} &\leftarrow \texttt{in\_forum(bob), in\_forum(connie), subject(soccer)} \end{aligned}$$

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:

$$J(\mathcal{P}_1, \cdots, \mathcal{P}_n) = \bigcup_{\mathcal{P}_i \in \{\mathcal{P}_1, \cdots, \mathcal{P}_n\}} \Big(S_1(\mathcal{P}_i) \cup S_2(\mathcal{P}_i) \cup S_3(\mathcal{P}_i) \Big) \bigcup C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$$

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

for each integer $i \mid \exists$ a program $\mathcal{P}_i \in P$ **do**
(* S_1 and S_3 are created *)
$S_1 = \emptyset$; $S_2 = \emptyset$; $S_3 = \emptyset$;
for each atom $a \in Var$ **do**
let a_1 be a new literal; $a_1.setName(a.getName + \text{“}p\text{”} + int2string(i))$;
let a_2 be a new literal; $a_2.setName(a.getName + \text{“ }'\text{ ”} + \text{“}p\text{”} + int2string(i))$;
let r be a new rule; $r.setHead([a_1.getName])$; $r.setBody([not\ a_2.getName])$;
let s be a new rule; $s.setHead([a_2.getName])$; $s.setBody([not\ a_1.getName])$;
$S_1 = S_1 \cup \{r\} \cup \{s\}$;
let a_3 be a new literal; $a_3.setName(\text{“}s\text{”} + a.getName + \text{“}p\text{”} + int2string(i))$;
let a_4 be a new literal; $a_4.setName(\text{“}fail\text{”} + \text{“}p\text{”} + int2string(i))$;
let t be a new rule; $t.setHead([a_4.getName])$;
$t.setBody([not\ a_4.getName, a_3.getName, not\ a_1.getName])$;
let u be a new rule; $u.setHead([a_4.getName])$;
$u.setBody([not\ a_4.getName, a_1.getName, not\ a_3.getName])$;
$S_3 = S_3 \cup \{t\} \cup \{u\}$;
end for
(* S_2 is created*)
for each rule $r \in \mathcal{P}_i \mid r : a \leftarrow b_1, \cdots, b_n$ **do**
let l be a new literal; $l.setName(\text{“}s\text{”} + a.getName + \text{“}p\text{”} + int2string(i))$;
$LSet = \emptyset$;
for each $b_j \in Body(r)$
let l_j be a new literal; $l_j.setName(b_j.getName + \text{“}p\text{”} + int2string(i))$;
let $LSet = LSet \cup \{l_j\}$;
end for
let $L = SetToList(LSet)$;
let s be a new rule; $s.setHead([l.getName])$; $s.setBody(L)$;
$S_2 = S_2 \cup \{s\}$;
end for
(* $\Gamma(\mathcal{P}_i)$ is created*)
$\Gamma(\mathcal{P}_i) = S_1 \cup S_2 \cup S_3$;
end for
(* $C(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is created *)
$C = \emptyset$;
let f be a new literal; $f.setName(\text{“}fail\text{”})$;
for each atom $a \in Var$ **do**
for each integer $i \in [1, n]$ **do**
let $a_{\mathcal{P}_i}$ be a new literal; $a_{\mathcal{P}_i}.setName(a.getName + \text{“}p\text{”} + int2string(i))$;
for each integer $j \in [1, n]$ **do**
if $i \neq j$ **then**
let $a_{\mathcal{P}_j}$ be a new literal; $a_{\mathcal{P}_j}.setName(a.getName + \text{“}p\text{”} + int2string(j))$;
let r be a new rule; $r.setHead([f.getName])$;
$r.setBody([not\ f.getName, a_{\mathcal{P}_i}.getName, not\ a_{\mathcal{P}_j}.getName])$;
$C = C \cup \{r\}$;
end if
end for
end for
end for
(* $J(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ is created *)
$J = \emptyset$;
for each integer $i \in [1, n]$ **do**
$J = J \cup \Gamma(\mathcal{P}_i)$;
end for
$J = J \cup C$;
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 $\mathrm{JFP}(\mathcal{P}_1, \cdots, \mathcal{P}_n)$. 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 $\mathrm{MJFP}(\mathcal{P}_1, \cdots, \mathcal{P}_n)$ 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)