Modular Analysis of Suspension Free cc Programs*

Enea Zaffanella
Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
zaffanel@di.unipi.it

Abstract

Compositional semantics allow to reason about programs in an incremental way, thus providing the formal base for the development of modular data-flow analyses. The major drawback of these semantics is their complexity, which increases as the interactions between different program modules become more sophisticated. As a consequence, the benefits of a compositional approach can be easily overcome by the incurred computational overhead. This observation applies in particular to concurrent constraint (cc) program modules, which can interact in very complicated ways by means of their synchronization primitive (the ask guard). In this work we provide a modular abstract interpretation framework for cc programs that partially solves this problem. By assuming the whole program suspension free, we show that it is possible to compositionally compute an approximation of its run-time behavior while keeping complexity under control. To this end we translate the cc program into a clp program and analyze the latter. Correctness results are easily stated in an implicative form. The approach is truly incremental and it does not depend on the chosen program decomposition; in particular no hypotheses are needed on the suspension freeness of each single module.

1 Introduction

Modular development is essential for the successful construction of large programs. Modularity imposes several requirements on the tools and techniques used for program development. In particular, it requires tools which behaves nicely with respect to the (explicit or implicit) program composition operators of the language. The notion of *compositional semantics* emerges from the need of providing a semantic support to modularity, since these semantics allow to express the meaning of a program in terms of the meaning of its constituent parts. In this paper we consider the problem of providing a compositional abstract interpretation framework for the analysis of *concurrent constraint* (cc) programs, the concurrent extension of constraint logic programs (clp) defined by Saraswat et al. [15, 14]. In particular, here we are interested in the compositionality with respect to the *union* of cc programs. Consider a cc program P which is split into modules $M_1, \ldots, M_n$, each module M_i having being analyzed separately so to obtain the information S_i on its run-time

*This work has been supported by the "PARFORCE" (Parallel Formal Computing Environment) BRA-Esprit II Project n. 6707.

behavior. In a truly compositional analysis framework, the information on the run-time behavior of the program P can be obtained by suitably composing $S_1, \ldots, S_n$. This has the big advantage that, should M_k be changed into M'_k for some reason, the information on the new overall program P' can be obtained by (1) analyzing only the changed module M'_k obtaining S'_k, and (2) combining S'_k with the old $S_1, \ldots S_{k-1}, S_{k+1}, \ldots S_n$, thus avoiding to re-analyze the unchanged modules. This is possible, of course, only if our analyzer is based on a compositional abstract interpretation, i.e. both the concrete and the abstract semantics are compositional. However, we have no *a priori* guarantee on the effectiveness of the approach. It might well be the case that our potential advantages (e.g., savings on the analysis time) are destroyed by the overhead imposed by the need of a compositional semantics. In our opinion, indeed, such an approach is very likely to be uneffective in the cc case, as a compositional semantics for these languages seems to be too complex for a practical use. This is because cc agents can interact in a subtle and complicated way by means of their synchronization mechanism and therefore easily introduce very strong dependencies between different modules.

Things are much better if we restrict ourselves to cc programs which are *suspension free*, meaning that their terminating computations do not contain suspended processes. This reactive property of cc programs is important for several reasons. First of all suspended computations usually correspond to undesired behaviors of the program; their presence is very likely to denote a programming error and, indeed, a lot of research work has been directed to the development of tools for proving a program suspension free [3, 4, 7]. Moreover, suspension freeness does not behave well with respect to compositionality, as it can be shown that by combining suspension free program modules we can produce suspended computations. These considerations suggest a *two steps* approach to the abstract interpretation of cc programs. In the first step the program is proved suspension free. The second step *assumes* suspension freeness and uses this information to perform the static analysis in a more efficient and precise way. In this paper we show how to perform the second step of the outlined approach in a modular *and* effective way, i.e. avoiding the complexity problems stated before. For the purposes of program analysis, as shown in [17], a suspension free cc program P can be safely translated into a clp program P' by removing all synchronizations (i.e. ask guards). The abstract interpretation of P', computed within any framework for clp, yields a correct approximation of the semantics of the original program P. Following [9], we will introduce a semantics for the translated programs which is compositional wrt the union of programs. This semantics is then approximated by applying the theory of abstract interpretation. The correctness result do not depend on the program decomposition we have chosen; in particular, it holds even if all the modules are not suspension free. Another important point to stress is that the *two steps* do not need to be chronologically ordered; the compositional abstract interpretation can be performed even *before* proving suspension freeness, thus allowing a truly incremental approach to analysis.

2 Constraint Systems and cc Languages

Constraint systems are semantic domains formalizing the gathering and the management of partial information.

Definition 2.1 (partial information system) [16]
A partial information system *is a quadruple* $\langle D, \Delta, Con, \vdash \rangle$ *where* D *is a denumerable set*

of elementary assertions (tokens), $\Delta \in D$ is a distinguished assertion (the least informative token), Con is a family of finite *subsets of D (the consistent subsets of tokens) and $\vdash \subseteq Con \times Con$ is the entailment relation satisfying (for $u, v, w \in Con$) $\emptyset \vdash \{\Delta\}$, $u \vdash v$ if $v \subseteq u$ and $u \vdash w$ if $(u \vdash v \wedge v \vdash w)$.*

Entailment closed sets of tokens are called *constraints* and provide representatives for the equivalence classes induced by the entailment relation; in particular, *true* denotes the set of all the trivial tokens. Note that, in a partial information system, all constraints are consistent. In order to model inconsistent information, we add a special token ∇ to D and say that, for all $P \in D$, the entailment $\{\nabla\} \vdash \{P\}$ holds; we also take $Con = \{u \subseteq D \mid card(u) < \omega\}$. Therefore, the entailment closure of $\{\nabla\}$ is the whole set D, which is denoted *false*. The *simple constraint system* generated by a partial information system is the complete lattice $\langle C, \dashv, true, false, \sqcup, \sqcap \rangle$ of all the constraints together with the partial order induced by the reverse of the entailment relation (which we will denote $\dashv$). We write $\sqcup$ to denote the constraint composition operator (the *lub* of the constraint system). Cylindric constraint systems [14] (constraint systems for short) are simple constraint systems in which the notion of variable is explicitly taken into account. The constraint algebra is enriched by diagonal elements d_{xy} and cylindric operators $\exists_x$ [13].

Definition 2.2 (constraint system)
A (cylindric) constraint system *$\mathcal{C} = \langle C, \dashv, true, false, \sqcup, \sqcap, \exists_x, d_{xy} \rangle_{x,y \in V}$ is an algebraic structure where $\langle C, \dashv, true, false, \sqcup, \sqcap \rangle$ is a simple constraint system; V is a denumerable set of variables; $\forall x, y \in V$, $\forall c, c' \in C$ we have $\exists_x false = false$, $\exists_x c \dashv c$, $c \dashv c'$ implies $\exists_x c \dashv \exists_x c'$, $\exists_x(c \sqcup \exists_x c') = \exists_x c \sqcup \exists_x c'$, $\exists_x(\exists_y c) = \exists_y(\exists_x c)$; $\forall x, y, z \in V$, $\forall c \in C$ we have $d_{xx} = true$, $z \not\equiv x, y$ implies $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$, $x \not\equiv y$ implies $c \dashv d_{xy} \sqcup \exists_x(c \sqcup d_{xy})$.*

We refer to [16] and [14] for a more detailed presentation. In the following, we write $\hat{x}$ to denote a sequence of distinct variables; we often write $\hat{x}$ to denote the *set* of variables occurring in the sequence $\hat{x}$. Given $\hat{x} = (x_1, \ldots, x_n)$ we write $\exists_{\hat{x}} c$ to denote the constraint $\exists_{x_1}(\ldots \exists_{x_n}(c))$; given $\hat{y} = (y_1, \ldots, y_n)$, we write $d_{\hat{x}\hat{y}}$ to denote the constraint $d_{x_1 y_1} \sqcup \ldots \sqcup d_{x_n y_n}$. When it is not specified in the context, we always assume that the arities of such tuples are matching. The projection operator $\overline{\exists}_{\hat{x}}$ is defined as $\overline{\exists}_{\hat{x}} c = \exists_{V \setminus \hat{x}}\, c$, i.e. it hides the information of all the variables but $\hat{x}$. The set of variables that *occurs free* in a constraint c is $vars(c) = \{x \in V \mid \exists_x c \neq c\}$.

As an example, let us formalize the constraint system of dependency relations, which can be used to propagate interesting properties of program variables (e.g. definiteness).

Example 2.1 [Dependency relations][3]
Let V be a finite set of variables and π be a property. The set of tokens is defined as $D = (\wp(V) \times \wp(V)) \cup \{\nabla\}$; the interpretation of a token $(X, Y) \in D$ is that if all the variables in Y satisfy property π, then all the variables in X satisfy the property too. The entailment $\models$ is defined (accordingly to the intuitive meaning of a set of tokens) as the minimal entailment relation on D such that, for all $R \subseteq D$ and $X, Y, W, Z \subseteq V$, if $X \subseteq Y$ then $\emptyset \models \{(X, Y)\}$, if $(R \models \{(X, Y)\} \wedge R \models \{(Y, Z)\})$ then $R \models \{(X, Z)\}$, if $(R \models \{(X, Y)\} \wedge R \models \{(W, Z)\})$ then $R \models \{(X \cup W, Y \cup Z)\}$. Given a set of tokens $R \subseteq D$, let $\varrho(R)$ denote its entailment closure. Moreover we can define $\exists_x R = \varrho\,(\varrho(R) \setminus \{(X, Y) \mid x \in X \cup Y\})$ and $d_{xy} = \varrho\,(\{(\{x\}, \{y\}), (\{y\}, \{x\})\})$. □

In the following we systematically confuse a set of tokens and the constraint generated by this set, that is its entailment closure.

2.1 Modeling nondeterminism

Each element in the domains of [14] models the result of a single computation (using the terminology of [11], each element of such domains is a *simple* constraint). In order to characterize the nondeterministic choice of (concurrent) constraint languages, *sets* of computations are considered in all the semantic definitions given in the literature.

In this paper, following [11], we adopt a more general solution by considering an algebraic "solution merging" operator $\oplus$. Let $\rho : \wp(C) \to \wp(C)$ be an upper closure operator on $\wp(C)$; define $\mathcal{P}_\rho(C) = \{\rho(S) \mid S \in \wp(C)\}$.

Definition 2.3 (disjunctive lifting)
The disjunctive lifting *of the constraint system* $\mathcal{C} = \langle C, \dashv, \mathit{true}, \mathit{false}, \sqcup, \sqcap, \exists_x, d_{xy} \rangle_{x,y \in V}$ *wrt* ρ *is defined as* $\mathcal{P}_\rho(\mathcal{C}) = \langle \mathcal{P}_\rho(C)), \subseteq, \mathbf{0}, \mathbf{1}, \oplus, \cap, \otimes, \boxminus_x, \delta_{xy} \rangle_{x,y \in V}$*, where* $\langle \mathcal{P}_\rho(C), \subseteq, \mathbf{0}, \mathbf{1}, \oplus, \cap \rangle$ *is a complete lattice;* $\mathbf{0} = \rho(\emptyset)$*,* $\mathbf{1} = \rho(C) = C$ *and* $\delta_{xy} = \rho(\{d_{xy}\})$*; for all* $S, S' \in \mathcal{P}_\rho(C)$ *we have* $S \otimes S' = \rho(\{c \sqcup c' \mid c \in S, c' \in S'\})$*,* $S \oplus S' = \rho(S \cup S')$ *and* $\boxminus_x(S) = \rho(\{\exists_x c \mid c \in S\})$*. The lifting function* $\{\!|\cdot|\!\} : C \to \mathcal{P}_\rho(C)$ *is defined as* $\{\!|c|\!\} = \rho(\{c\})$*.*

Note that the (lifted) composition operator $\otimes$ is associative and commutative, but in general it is not idempotent. By taking $\rho = id$, we obtain the powerset of the constraint system $\mathcal{C}$, where $\oplus = \cup$, and the other operators on $\mathcal{P}_{id}(C)$ (namely $\otimes$, $\boxminus_x$ and δ_{xy}) are simply the linear extensions of the corresponding operators on C. This domain can be used to model computed answer constraints and indeed it is at the base of the extension to the clp case of the s-semantics construction defined in [8] for pure logic programs. It is worth to stress that, in the general case, the disjunctive lifting of a constraint system is not itself a constraint system, as the lifted operators do not necessarily satisfy the axioms of Definition 2.2. In the following, when no confusion can arise, we freely omit the subscript ρ; moreover, depending on the context, the symbol c will be used to denote an element ranging both on the set of constraints C and on the set of disjunctive elements $\mathcal{P}(C)$.

2.2 The Language

In order to simplify the program translation defined in subsection 3.1, we introduce concurrent constraint languages [15, 14] by giving a syntax based on clauses; each concurrent process is thus defined by a set of clauses. Each clause consists of three parts: a head, a (ask and tell) pair of constraints and a body part, which is a sequence of process calls representing their parallel composition (see Table 1). Given a body $B = p_1(\hat{x}_1), \ldots, p_n(\hat{x}_n)$ we define the *variables tuple* of B as $vt(B) = \hat{x}_1 \cdot \ldots \cdot \hat{x}_n$; then, given a clause $cl \equiv p_0(\hat{x}_0) :- a : c \,\square\, B$, we define $vt(cl) = \hat{x}_0 \cdot vt(B)$. We consider only clauses in *normal form.*

Definition 2.4 (normal form)
A cc clause $cl \equiv h :- a : c \,\square\, B$ *is in* normal form *iff the variables in* $vt(cl)$ *are all distinct,* $vars(a) \subseteq vt(cl)$ *and* $vars(c) \subseteq vt(cl)$*.*

All the variables that do not occur in the head (namely, all the variables in the body B) are considered *local* variables (this is needed in order to preserve the referential transparency property). Note that we are not restricting the expressive power of the language as any cc clause can be easily mapped to an equivalent clause in normal form.

Program	::=	ϵ | Clause Program
Clause	::=	Head :− Ask : Tell $\Box$ Body.
Body	::=	ϵ | Call Body
Head, Call	::=	$p(\hat{x})$
Ask, Tell	::=	c

Table 1: The syntax

The operational model is described by a transition system $T = (\mathit{Conf}\,, \xrightarrow{W}_P)$, where $\mathit{Conf} = \{\langle\, c \,\Box\, B \,\rangle \mid c \in C,\ B \in \text{Body}\}$ and the transition relation $\xrightarrow{W}_P$ is defined by the following transition rule:

$$\frac{cl \ll_X P \qquad a \dashv (d_{\hat{x}\hat{y}} \sqcup \sigma)}{\langle\, \sigma \,\Box\, B_1, p(\hat{x}), B_2 \,\rangle \ \xrightarrow{W}_P \ \langle\, c \sqcup d_{\hat{x}\hat{y}} \sqcup \sigma \,\Box\, B_1 \cdot B' \cdot B_2 \,\rangle}$$

where $cl \equiv p(\hat{y})\,{:-}\,a : c \,\Box\, B'$ and $X = W \cup \mathit{vars}(\sigma) \cup \mathit{vt}(B_1, p(\hat{x}), B_2)$.

Note that $\xrightarrow{W}_P$ is indexed on the program P and on the set of variables of interest $W \subseteq V$, which are the variables occurring in the initial configuration. When no confusion can arise we simply write $\xrightarrow{W}$, thus omitting the program index P. In a configuration $\langle\, \sigma \,\Box\, B \,\rangle \in \mathit{Conf}$, the (finite) constraint $\sigma \in C$ is the global store while the body B represents the residual computation. The reduction of a process call $p(\hat{x})$ implements nondeterministic choice, parameter passing, synchronization and constraint publication all at once. We nondeterministically select a program clause $cl \equiv p(\hat{y})\,{:-}\,a : c \,\Box\, B'$ which is *renamed apart* wrt both the current configuration and the set of variables of interest W (notation $\mathit{obj} \ll_X S$ means that the object obj is obtained by choosing an element of S and renaming it so that it contains none of the variables in X). Then we check whether the ask constraint a is entailed by the current store σ augmented with the parameter passing constraint $d_{\hat{x}\hat{y}}$. If this is the case then the computation commits, i.e. the call $p(\hat{x})$ is replaced by the clause body B' and the global store is augmented by the parameter passing information and the tell constraint c. Note that we do not perform any consistency check on the tell constraint. Otherwise, if no such clause exists, the process call cannot be reduced and we say that $p(\hat{x})$ is *stuck* in the current configuration. A computation s for a program P and an initial configuration $\langle\, \sigma \,\Box\, B \,\rangle$ is a possibly infinite sequence of configurations $\{\,\langle\, c_i \,\Box\, B_i \,\rangle\,\}_i$ such that $B_0 = B$ and $c_0 = \sigma$ and for all $i < |s|$, $\langle\, c_i \,\Box\, B_i \,\rangle \xrightarrow{W} \langle\, c_{i+1} \,\Box\, B_{i+1} \,\rangle$. Let $\not\longrightarrow$ denote the absence of admissible transitions. Sequences reaching configurations $\langle\, c_n \,\Box\, B_n \,\rangle$ such that $\langle\, c_n \,\Box\, B_n \,\rangle \not\longrightarrow$ are called *terminating* computations and $c_n \in C$ is the (finite) computed answer constraint. A terminating computation such that $B_n = \epsilon$ is a *successful* computation, otherwise ($B_n \neq \epsilon$) it is a *suspended* computation.

Definition 2.5 (finite semantics) *Let $G = \langle\, \sigma \,\Box\, B \,\rangle$ and let $W = \mathit{vars}(\sigma) \cup \mathit{vt}(B)$. The* finite semantics *for program P and configuration G is*

$$\mathcal{O}[\![\, P \,]\!](G) = \bigoplus \left\{ \{\!|\, \exists_W c \,|\!\} \;\middle|\; G \ (\xrightarrow{W}_P)^* \ \langle\, c \,\Box\, B' \,\rangle \not\longrightarrow_P \right\}$$

Note that the computed answer constraints are projected onto the set of variables of interest, namely the variables occurring in the initial configuration G. Moreover, the finite semantics considers the results of all the terminating computations, regardless of whether they are successful or suspended. We say that program P and configuration G are *suspension free* if there is no suspended computation for P starting from G.

3 Compositional Analysis: a Compromise

In many contexts, e.g. for the designing and analysis of complex concurrent programs, adopting a compositional approach is a must. Unfortunately, it is known that the suspension freeness of a program is not compositional, namely we can obtain suspended computations by combining suspension free program chunks.

Example 3.1 Consider the following programs.

$$P1 : \begin{cases} p(x) :- \mathit{true} : x = a \,\Box\, . \\ r(x) :- \mathit{true} : x = y \,\Box\, q_1(y,z). \\ q_1(y,z) :- y = a : \mathit{true} \,\Box\, . \\ q_1(y,z) :- z = a : \mathit{true} \,\Box\, . \end{cases} \qquad P2 : \begin{cases} p(x) :- \mathit{true} : x = b \,\Box\, . \\ r(x) :- \mathit{true} : x = y \,\Box\, q_2(y,z). \\ q_2(y,z) :- y = b : \mathit{true} \,\Box\, . \\ q_2(y,z) :- z = b : \mathit{true} \,\Box\, . \end{cases}$$

Given the initial configuration $G = \langle\, x = y \,\Box\, p(x), r(y)\,\rangle$, it can be easily observed that $P1$ (resp. $P2$) and G are suspension free, while for $P = P1 \cup P2$ and G we have the following suspended computation (where we have renamed variables and projected the store for readability):

$$\begin{aligned} \langle\, x_1 = x_2 \,\Box\, p(x_1), r(x_2)\,\rangle \quad & \xrightarrow{W}_P \quad \langle\, x_1 = x_2 = a \,\Box\, r(x_2)\,\rangle \\ & \xrightarrow{W}_P \quad \langle\, x_1 = x_2 = a \,\Box\, q_2(x_2, z)\,\rangle \not\rightarrow_P \end{aligned}$$

$\Box$

The viceversa also holds, i.e. it is possible to obtain a suspension free program by combining two or more program chunks which are not suspension free.

Example 3.2 Consider the following programs.

$$P1 : \begin{cases} p(x,y) :- \mathit{true} : x = a \,\Box\, . \\ q(x,y) :- y = b : \mathit{true} \,\Box\, . \end{cases} \qquad P2 : \begin{cases} p(x,y) :- \mathit{true} : y = b \,\Box\, . \\ q(x,y) :- x = a : \mathit{true} \,\Box\, . \end{cases}$$

Given the initial configuration $G = \langle\, x_1 = x_2, y_1 = y_2 \,\Box\, p(x_1, y_1), q(x_2, y_2)\,\rangle$, it can be easily observed that both $P1$ and $P2$ with configuration G are not suspension free (they have no successful computation at all), while for $P = P1 \cup P2$ and G we can easily prove suspension freeness. $\Box$

Hence, even if we are able to prove the suspension freeness of all the program modules, the whole program is not proved to be suspension free. Obviously, a semantics which is compositional wrt the union of cc programs can be defined, but it is very likely that, in order to be correct, this semantics would have to encode the whole computation structure.

$NoSynch[\,\epsilon\,] \;=\; \epsilon$
$NoSynch[\,cl\ progr\,] \;=\; NoSynch[\,cl\,]\ NoSynch[\,progr\,]$
$NoSynch[\,p(\hat{x}) :\!-\, a : c \,\square\, G.\,] \;=\; p(\hat{x}) :\!-\, (a \sqcup c) \,\square\, G.$

Table 2: The transformation *NoSynch*

3.1 The *two-steps* approach

These considerations suggest a two step approach to the modular abstract interpretation of cc programs. In the first step we prove that the complete program is suspension free. The second step *assumes* the suspension freeness and uses this information to perform the static analysis in a more efficient way. Following [17], we now define a semantics for cc programs that observes *successful* computations only.

Definition 3.1 (success semantics) *Let* $G = \langle\, \sigma \,\square\, B \,\rangle$ *and let* $W = vars(\sigma) \cup vt(B)$*. The* success semantics *for program* P *and configuration* G *is*

$$\mathcal{SS}[\![\, P \,]\!](G) = \bigoplus \left\{ \{\!|\, \exists_W c \,|\!\} \;\middle|\; G \ (\xrightarrow{W}_P)^* \ \langle\, c \,\square\, . \,\rangle \right\}$$

Remark 3.1 *If* P *and* G *are suspension free then* $\mathcal{O}[\![\, P \,]\!](G) = \mathcal{SS}[\![\, P \,]\!](G)$*.*

Hence, if our program is suspension free, we can use the success semantics for the correctness proofs of our abstract interpretation framework. The success semantics for cc programs can be easily approximated by defining a syntactic transformation called *NoSynch* [17], mapping all the ask constraints of the program into equivalent tell constraints, i.e. removing all the synchronization tests. As pointed out in [17], the transformed program is a *sequential* constraint logic program (plus syntactic sugar), thus allowing the application of the generalized approach as developed in [11]. To make this observation more evident, in this work we redefine *NoSynch* as a translation, directly mapping cc programs into clp programs (see Table 2); note that, while translating the clauses, we also map each constraint $a \sqcup c \in C$ to its disjunctive lifting $\{\!|a \sqcup c|\!\} \in \mathcal{P}(C)$. By lifting the function *vars* on $\mathcal{P}(C)$ (i.e. by defining $vars(c) = \{x \in V \mid \exists_x c \neq c\}$), it can be easily proved that *NoSynch* preserves the normal form of clauses.

3.2 The semantics of Ω-programs

Definition 3.2 (Ω-program) [1, 9] *An* Ω*-program is a (concurrent) constraint logic program* P *together with a set* Ω *of predicate symbols.*

Let $pred(P)$ denote the set of predicates that are defined in P (this set can be a proper *subset* of the predicates occurring in P). A predicate symbol occurring in Ω is considered to be only partially defined in P, i.e. it is *open* in P. On the other hand, predicate symbols occurring in P but not in Ω have to be considered *closed*, i.e. fully specified[1]. This means that an Ω-program P can be composed with other programs provided that these ones do not further specify the predicates which are closed in P.

[1] In order to avoid technical problems when defining the compositional semantics, we assume that all the closed predicates are defined in P; namely, if $p \in \Pi \setminus \Omega$ occurs in P then $p \in pred(P)$. This condition can always be met by eventually adding a dummy definition $p(\hat{x}) :\!-\, \mathbf{0} \,\square\,$.

Definition 3.3 (Ω-union) [1, 9] *Let P_1 be an Ω_1-program and P_2 be an Ω_2-program. If $\Omega \subseteq \Omega_1 \cap \Omega_2$ and $(pred(P_1) \cap pred(P_2)) \subseteq (\Omega_1 \cap \Omega_2)$ then $P_1 \cup_\Omega P_2$ is the Ω-program $P_1 \cup P_2$. Otherwise $P_1 \cup_\Omega P_2$ is not defined.*

From now on, we consider a clp program P obtained by applying the translation *NoSynch*. The key observation of [1] is that compositionality wrt program union can be achieved by considering interpretations based on (possibly non-unit) *clauses*. Hence, the compositional *fixpoint* semantics $\mathcal{F}_\Omega(P)$ is obtained by extending the semantic construction of the previous section.

Definition 3.4 (clause skeleton)
Let $cl : p_0(\hat{x}_0) :- c \,\Box\, p_1(\hat{x}_1), \ldots, p_n(\hat{x}_n)$ be a clp clause in normal form. The skeleton of cl is defined as $skel(cl) = \langle p_0, p_1, \ldots, p_n \rangle \in \Pi^+$. Given a set $\Omega \subseteq \Pi$ of predicate symbols, the set of Ω-skeletons is defined as $\Pi\Omega^* = \{\langle p_0, \ldots, p_n \rangle \mid p_0 \in \Pi,\ \langle p_1, \ldots, p_n \rangle \in \Omega^* \}$.

Definition 3.5 (Ω-interpretation) *Let $cl \equiv p_0(\hat{x}_0) :- c \,\Box\, B$ and $cl' \equiv p'_0(\hat{y}_0) :- c' \,\Box\, B'$ be two clauses in normal form. We write $cl \preceq cl'$ iff $skel(cl) = skel(cl')$ and $\exists_{\hat{x}}(\delta_{\hat{x}\hat{z}} \otimes c) \subseteq \exists_{\hat{y}}(\delta_{\hat{y}\hat{z}} \otimes c')$, where $\hat{x} = vt(cl)$, $\hat{y} = vt(cl')$ and $\hat{z}$ is a tuple of distinct variables such that $\hat{x} \cap \hat{z} = \hat{y} \cap \hat{z} = \emptyset$. Then $cl \sim cl'$ iff both $cl \preceq cl'$ and $cl' \preceq cl$ hold. The interpretation base is $\mathcal{B}^\Pi_\Omega = \{ [cl]_\sim \mid skel(cl) \in \Pi\Omega^* \}$. An Ω-interpretation $I \in \Im^\Pi_\Omega$ is any subset of $\mathcal{B}^\Pi_\Omega$ such that for each $sk \in \Pi\Omega^*$ there exists one and only one clause $cl \in I$ such that $skel(cl) = sk$.*

As in the previous section, we omit the superscript Π when no confusion can arise and we order Ω-interpretations by extending the partial order $\preceq$ defined on clauses. The domain $\langle \Im_\Omega, \preceq \rangle$ is again a complete lattice.

Let us introduce the merge operator $(\cdot)^\flat : \wp(\mathcal{B}_\Omega) \to \Im_\Omega$.

$$(S)^\flat = \left\{ [cl]_\sim \;\middle|\; \begin{array}{l} \exists\ sk \in \Pi\Omega^* \text{ such that} \\ cl \equiv h :- c \,\Box\, B,\ skel(cl) = sk,\ \hat{x} = vt(cl) \\ c = \bigoplus \left\{ \exists_{\hat{y}}(\delta_{\hat{y}\hat{x}} \otimes c') \;\middle|\; \begin{array}{l} cl' \equiv h' :- c' \,\Box\, B' \ll_{\hat{x}} S \\ skel(cl') = sk,\ \hat{y} = vt(cl') \end{array} \right\} \end{array} \right\}$$

The *lub* $\uplus$ on $\Im_\Omega$ is defined as before: $\biguplus_{i \in J} I_i = \left(\bigcup_{i \in J} I_i \right)^\flat$.

Given the set of predicates Ω, let $Id_\Omega = \{ [p(\hat{x}) :- \delta_{\hat{y}\hat{x}} \,\Box\, p(\hat{y}).]_\sim \mid p \in \Omega \}$. The T^Ω_P operator is obtained by unfolding the program P wrt the Ω-interpretation I augmented with Id_Ω. A careful use of clauses renaming and cylindric operators allows to preserve normal forms.

Definition 3.6 (T^Ω_P) [9]
Let $cl_0 \equiv p_0(\hat{x}_0) :- c_0 \,\Box\, B$ be a clp clause, where $B = p_1(\hat{x}_1), \ldots, p_n(\hat{x}_n)$; let I be an Ω-interpretation and $X_0 = \emptyset$. We define the operator $T^\Omega_{cl_0} : \Im_\Omega \to \wp(\mathcal{B}_\Omega)$ as follows:

$$T^\Omega_{cl_0}(I) = \left\{ [p_0(\hat{x}_0) :- c \,\Box\, B_1, \ldots, B_n]_\sim \;\middle|\; \begin{array}{l} \forall\, i \in \{1, \ldots, n\}\ .\ X_i = X_{i-1} \cup vt(cl_{i-1}) \\ \exists\ cl_i \equiv p_i(\hat{y}_i) :- c_i \,\Box\, B_i \ll_{X_i} (I \cup Id_\Omega) \\ c = \exists_{vt(B)} \left(c_0 \otimes \left(\bigotimes_{i=1}^{n} \exists_{\hat{y}_i}(\delta_{\hat{y}_i\hat{x}_i} \otimes c_i) \right) \right) \end{array} \right\}$$

Thus $T^\Omega_P : \Im_\Omega \to \Im_\Omega$ is defined as $T^\Omega_P(I) = \left(\bigcup \{ T^\Omega_{cl}(I) \mid cl \in P \} \right)^\flat$ and $\mathcal{F}_\Omega(P) = T^\Omega_P \uparrow \omega$.

If $\Omega \neq \emptyset$ then any Ω-interpretation is an infinite set, as the set of Ω-skeletons $\Pi\Omega^*$ is infinite. For simplicity, when denoting an Ω-interpretation we usually drop all the clauses $h :- c \,\square\, B$ such that $c = \mathbf{0}$. Even by adopting this representation convention, in general Ω-interpretations are infinite sets (we will discuss this point later in subsection 4.1). As a special case, by taking $\Omega = \emptyset$, we obtain the set of $\emptyset$-interpretations $\Im_\emptyset = \Im$ which is exactly the set of interpretations defined in the previous section; moreover, $T_P^\emptyset = T_P$ and $\mathcal{F}_\emptyset = \mathcal{F}$.

Theorem 3.1 (compositionality) [9] *Let P_1 be an Ω_1-program, P_2 be an Ω_2-program and let $P_1 \cup_\Omega P_2$ be defined. Then $\mathcal{F}_\Omega(\mathcal{F}_{\Omega_1}(P_1) \cup_\Omega \mathcal{F}_{\Omega_2}(P_2)) = \mathcal{F}_\Omega(P_1 \cup_\Omega P_2)$.*

We have seen that the compositional semantics of an Ω-program P encodes the dependences of P from the *open* predicates Ω. By giving the semantics of the predicates in Ω we fully specify the semantics of P; for this reason we call Ω-closure any interpretation $I \in \Im_\emptyset^\Omega$. Given the Ω-programs P_1 and P_2, we define the equivalence relation $\approx_\Omega$ that holds whenever P_1 and P_2 are equivalent under all the possible Ω-closures (here we regard an Ω-closure as an Ω-program).

Definition 3.7 *$P_1 \approx_\Omega P_2$ iff for all Ω-closures I we have $\mathcal{F}(P_1 \cup_\emptyset I) = \mathcal{F}(P_2 \cup_\emptyset I)$.*

Proposition 3.2 [1] *$P_1 \approx_\Omega \mathcal{F}_\Omega(P_1)$ and $(\mathcal{F}_\Omega(P_1) = \mathcal{F}_\Omega(P_2) \;\Rightarrow\; P_1 \approx_\Omega P_2)$.*

4 Correctness Results

In this section we formalize the notion of *correctness* of an abstract domain wrt (the disjunctive lifting of) a constraint system. This condition implies the correctness of the (compositional) abstract semantic construction, which is obtained my mimicking the concrete one on the abstract domain. The definitions are inspired by [11, 12], which in turn are based on the classical works on abstract interpretation theory [5, 6].

Definition 4.1 *An abstract domain $\mathcal{A} = \langle L, \sqsubseteq^\sharp, \mathbf{0}^\sharp, \mathbf{1}^\sharp, \oplus^\sharp, \sqcap^\sharp, \otimes^\sharp, \exists_x^\sharp, \delta_{xy}^\sharp \rangle_{x,y \in V}$ is a complete lattice $\langle L, \sqsubseteq^\sharp, \mathbf{0}^\sharp, \mathbf{1}^\sharp, \oplus^\sharp, \sqcap^\sharp \rangle$ together with a binary operator $\otimes^\sharp$, a family of unary operators $\exists_x^\sharp$ for $x \in V$ and a family of distinguished elements $\delta_{xy}^\sharp \in L$ for $x, y \in V$.*

The safeness of the overall construction can be achieved by suitably relating the concrete and abstract elements and by imposing correctness conditions regarding the domain's operators.

Definition 4.2 *An abstract domain $\mathcal{A} = \langle L, \sqsubseteq^\sharp, \mathbf{0}^\sharp, \mathbf{1}^\sharp, \oplus^\sharp, \sqcap^\sharp, \otimes^\sharp, \exists_x^\sharp, \delta_{xy}^\sharp \rangle_{x,y \in V}$ is* correct *wrt the concrete domain $\mathcal{P}(\mathcal{C}) = \langle \mathcal{P}(C), \subseteq, \mathbf{0}, \mathbf{1}, \oplus, \cap, , \otimes, \exists_x, \delta_{xy} \rangle_{x,y \in V}$ using α iff there exists an upper Galois insertion (α, γ) relating $\mathcal{P}(C)$ and L and $\forall c, c' \in \mathcal{P}(C)$, $\forall x, y \in V$ $\alpha(c \otimes c') \sqsubseteq^\sharp \alpha(c) \otimes^\sharp \alpha(c')$, $\alpha(\exists_x c) \sqsubseteq^\sharp \exists_x^\sharp \alpha(c)$ and $\alpha(\delta_{xy}) \sqsubseteq^\sharp \delta_{xy}^\sharp$.*

Example 4.1 [groundness][11]
Consider the Herbrand constraint system (together with the disjunctive lifting induced by $\rho = id$) and consider the abstract domain induced by the constraint system of dependency relations defined in Example 2.1. It is known that the latter is equivalent to the domain *Def* of definite and positive boolean functions defined on variables V, together with the element *False*. We now relate the two domains by exploiting the groundness dependency information. Let $c \equiv \exists_Y \{x_i = t_i \mid i = 1, \ldots, n\} \in H$ be a Herbrand constraint in solved

form[2] and, for all $i \in \{1,\dots,n\}$, let $W_i = \mathit{vars}(t_i)$. Define the function $\alpha_H : H \to \mathit{Def}$ such that $\alpha(c) = \exists_Y\,(\wedge_{i=1}^{n}(x_i \leftrightarrow \wedge W_i))$ and let $\alpha_H(\mathit{false}) = \mathit{False}$. The abstraction function $\alpha : \wp(H) \to \mathit{Def}$ is defined as the additive extension of α_H, i.e. $\alpha(S) = \oplus^\sharp\,\{\alpha_H(c) \mid c \in S\}$. Then the constraint system Def is correct wrt $\mathcal{P}(H)$ using α [11]. □

By substituting each occurrence of elements and operators of the concrete domain by the corresponding elements and operators of the abstract domain, we systematically derive the notions of abstract program, goal and clauses; abstract clauses are ordered by $\preceq^\sharp$ and we easily define the set $\Im^\sharp_\Omega$ of abstract Ω-interpretations, which is ordered, as before, by extending the relation $\preceq^\sharp$; then we define the abstract immediate consequences operator and the abstract bottom-up semantic function $\mathcal{F}^\sharp_\Omega$.

Theorem 4.1 (correctness) $(\alpha \circ \mathcal{F}_\Omega)(P) \preceq^\sharp (\mathcal{F}^\sharp_\Omega \circ \alpha)(P)$

We are now ready to formalize the relation between the semantics of a cc program and the semantics of its *NoSynch* translation, the latter semantics being computed compositionally.

Let P_1 and P_2 be two cc programs; let Ω_1 (resp. Ω_2) be the set of predicates that are open in P_1 (resp. P_2), such that $P = P_1 \cup_\emptyset P_2$ is defined. Given the cc configuration G, let $G^\sharp$ be its corresponding abstract clp goal and define $P_i^\sharp = \alpha(\mathit{NoSynch}(P_i))$, where $i = 1, 2$.

Corollary 4.2 $\alpha(\,\mathcal{SS}[\![\,P\,]\!](G)\,) \sqsubseteq^\sharp \mathcal{F}^\sharp_\emptyset(\,\mathcal{F}^\sharp_{\Omega_1}(P_1^\sharp) \cup_\emptyset \mathcal{F}^\sharp_{\Omega_2}(P_2^\sharp)\,)(G^\sharp)$

Corollary 4.3 *If program $P = P_1 \cup_\emptyset P_2$ and configuration G are suspension free then*

$$\alpha(\,\mathcal{O}[\![\,P\,]\!](G)\,) \sqsubseteq^\sharp \mathcal{F}^\sharp_\emptyset(\,\mathcal{F}^\sharp_{\Omega_1}(P_1^\sharp) \cup_\emptyset \mathcal{F}^\sharp_{\Omega_2}(P_2^\sharp)\,)(G^\sharp)$$

Obviously, these results can be generalized to the case of $n > 2$ program modules. Also note that they do not depend on the program decomposition we have chosen.

4.1 Termination

Up to now, we have described an abstract interpretation framework. To turn the abstract interpretation into a static analysis, we have to be sure that the approximation of the semantics is finitely computable. Note that, as the set $\Pi\Omega^*$ is infinite (unless $\Omega = \emptyset$), we can obtain a diverging computation even if our abstract domain is finite. A general way to overcome the problem is the definition of a widening operator on the lattice of abstract Ω-interpretations. Some proposals are in the literature; e.g., in [3] sequences of process calls in a cc configuration are handled by applying another approximation layer, called $*$-abstraction; the same technique has been applied in [2] for the compositional analysis of pure logic programs; this solution enforces termination but must pay in terms of accuracy. An interesting result has been achieved in [10] for the case of *finite* abstract domains. In this case a finite characterization of the compositional semantics of a (constraint) logic program can be obtained (without any loss of precision) as the result of a finite number of iterations of the immediate consequences operator T_P^Ω. In practice, instead of checking if we have reached the fixpoint of T_P^Ω, we check for the T-stability of the n-th iterate.

Theorem 4.4 [10]
Let P be an Ω-program computing on a finite constraint system. There exists an $n < \omega$ such that for all $m \geq n$ we have $(T_P^\Omega \uparrow n) \approx_\Omega (T_P^\Omega \uparrow m)$.

[2] Namely, $i \neq j$ entails $x_i \not\equiv x_j$, $Y = \bigcup_{i=1}^{n} \mathit{vars}(t_i)$ and $x_i \notin Y$ for all $i \in \{1,\dots,n\}$.

Note that having assumed the finiteness of the domain, we obtain the finiteness of the T-stability check also, as we have a finite number of possible Ω-closures each one being a finite object.

5 Example

To clarify the picture, we now show a compositional analysis computing the groundness dependencies between the variables of a cc program defined on the Herbrand constraint system. In the program chunk P, process q is left unspecified (as usual, we denote anonymous Herbrand variables by underscores).

$$P: \begin{cases} g(x,y) \,{:}{-}\, \mathit{true} : x = x_1 = z, y = y_1 \,\Box\, p(x_1,y_1), q(z). \\ p(x,y) \,{:}{-}\, x = [a|\_] : x = [a|x_1], y = [b|y_1] \,\Box\, p(x_1,y_1). \\ p(x,y) \,{:}{-}\, x = [] : y = [] \,\Box\, . \end{cases}$$

The corresponding abstract program is $P^\sharp = (\alpha \circ \mathit{NoSynch})(P)$, where we use the α defined in Example 4.1:

$$P^\sharp: \begin{cases} g(x,y) \,{:}{-}\, (x \leftrightarrow x_1 \leftrightarrow z) \wedge (y \leftrightarrow y_1) \,\Box\, p(x_1,y_1), q(z). \\ p(x,y) \,{:}{-}\, (x \leftrightarrow x_1) \wedge (y \leftrightarrow y_1) \,\Box\, p(x_1,y_1). \\ p(x,y) \,{:}{-}\, x \wedge y \,\Box\, . \end{cases}$$

Let $\Omega_1 = \{q\}$; then $\mathcal{F}^\sharp_{\Omega_1}(P^\sharp) = \{\, p(x,y) \,{:}{-}\, x \wedge y \,\Box\, .,\ g(x,y) \,{:}{-}\, x \wedge y \wedge z \,\Box\, q(z).\,\}$. Let us consider the initial configuration $G = \langle\, \mathit{true} \,\Box\, g(x,y)\,\rangle$ and let Q be any Ω_2-program such that $P \cup_\emptyset Q$ is defined. The informal reading of the previous abstract denotation is that all the *successful* computations of $P \cup_\emptyset Q$ starting from G bind both variables to ground terms. Indeed, let $\Omega_2 = \emptyset$ and consider the following two alternative Ω_2-programs defining the process q.

$$Q_1: \begin{cases} q(x) \,{:}{-}\, \mathit{true} : x = [a|x_1] \,\Box\, q(x_1). \\ q(x) \,{:}{-}\, \mathit{true} : x = [] \,\Box\, . \end{cases} \qquad Q_2: \begin{cases} q(x) \,{:}{-}\, \mathit{true} : x = [\_|x_1] \,\Box\, q(x_1). \\ q(x) \,{:}{-}\, \mathit{true} : x = [] \,\Box\, . \end{cases}$$

Program $P_1 = P \cup Q_1$ and goal G are suspension free and it can be easily observed that in all the answers of the finite semantics $\mathcal{O}[\![\, P_1 \,]\!](G)$ the variables x and y are ground, thus showing the correctness of the analysis. Things are different when considering program $P_2 = P \cup Q_2$, which is *not* suspension free; in this case the finite semantics contains terminating computations reaching final stores in which both x and y are not ground. Nonetheless, for all the *successful* computations, correctness still holds (there is only one successful computation for P_2 and G, yielding the store $(x = [], y = [])$).

6 Conclusions

Due to the sophisticated synchronization mechanism provided by ask guards, the complexity overhead of a compositional semantics for concurrent constraint programs can easily overcome the benefits of a modular approach to the static analysis of these languages. In this work we have provided a modular abstract interpretation framework that partially solves this problem by assuming that the complete cc program is suspension free. For the purposes of program analysis, a suspension free cc program can be safely translated into a clp program by removing all its ask guards. The abstract interpretation of the new program, computed within any framework for clp, yields a correct approximation of the semantics of the original program. The complexity of this construction is limited, as the translation greatly simplifies the interactions between the program modules. This methodology provides results in implicative form: all the computations of the complete program that do not suspend are correctly described by the compositionally computed approximation. The approach is truly incremental and it does not depend on the chosen program decomposition: no hypotheses are made on the suspension freeness of each program module.

References

[1] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *TCS*, 122(1-2):3–47, 1994.

[2] M. Codish, S. K. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *Proc. POPL'93*, pages 451–464. ACM Press, 1993.

[3] M. Codish, M. Falaschi, and K. Marriott. Suspension Analysis for Concurrent Logic Programs. In *Proc. ICLP'91*, pages 331– 345. The MIT Press, 1991.

[4] M. Codish, M. Falaschi, K. Marriott, and W. Winsborough. Efficient Analysis of Concurrent Constraint Logic Programs. In *Proc. ICALP'93*, *LNCS*, pages 633–644, 1993.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. POPL'77*, pages 238–252, 1977.

[6] P. Cousot and R. Cousot. Systematic Design of Program Analysis Frameworks. In *Proc. POPL'79*, pages 269–282, 1979.

[7] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence and Concurrent Constraint Programming. In *Proc. AMAST'95*, 1995.

[8] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A new Declarative Semantics for Logic Languages. In *Proc. ICLP'88*, pages 993–1005. The MIT Press, 1988.

[9] M. Gabbrielli, M.G. Dore, and G. Levi. Observable semantics for Constraint Logic Programs. *JLC*, 5(2):133–171, 1995.

[10] M. Gabbrielli, R. Giacobazzi, and D. Montesi. Modular logic programs over finite domains. In *Proc. GULP'93*, pages 663–678, 1993.

[11] R. Giacobazzi, S. K. Debray, and G. Levi. A Generalized Semantics for Constraint Logic Programs. In *Proc. FGCS'92*, pages 581–591, 1992.

[12] R. Giacobazzi, S.K. Debray, and G. Levi. Generalized Semantics and Abstract Interpretation for Constraint Logic Programs. *JLP*, 1995.

[13] L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras. Part I and II.* North-Holland, Amsterdam, 1971.

[14] V. A. Saraswat, M. Rinard, and P. Panangaden. Semantic Foundation of Concurrent Constraint Programming. In *Proc. POPL'91*, pages 333–353. ACM, 1991.

[15] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. POPL'90*, pages 232–245. ACM, 1990.

[16] D. Scott. Domains for Denotational Semantics. In *Proc. ICALP'82*, *LNCS*, pages 577–613. Springer-Verlag, 1982.

[17] E. Zaffanella, G. Levi, and R. Giacobazzi. Abstracting Synchronization in Concurrent Constraint Programming. In *Proc. PLILP'94*, *LNCS*, pages 57–72. Springer-Verlag, 1994.