An Assertion language for slicing Constraint Logic Languages

M. Falaschi[1] C. Olarte[2]

[1] Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche
Università di Siena, Italy.
moreno.falaschi@unisi.it.
[2] ECT, Universidade Federal do Rio Grande do Norte, Brazil
carlos.olarte@gmail.com.

Abstract. Constraint Logic Programming is a language scheme for combining two declarative paradigms: constraint solving and logic programming. Concurrent Constraint Programming (CCP) is a declarative model for concurrency where agents interact by telling and asking constraints (pieces of information) in a shared store. In a previous paper we have developed a framework for dynamic slicing of CCP. Slicing is useful for debugging. The main idea in dynamic slicing is that the user is able to recognize that a partial computation is wrong. Hence the user marks some parts of the final state (a subset of the constraints and processes), which correspond to the data and processes that she wants to emphasize and study more deeply. Then, an automatic process of slicing begins, and the partial computation is "depurated", by removing the information which is not relevant to compute the emphasized information. In this paper we extend the framework to Constraint Logic Programs, generalizing the previous work. Moreover, we make one step further in the direction of automatizing the slicing process. We provide an assertion language suitable for these languages, which allows the user to specify some properties of the computations in her program. If a state in a computation does not satisfy an assertion then some "wrong" information is identified and an automatic slicing process can start. We show that our framework can be integrated with the previous semi-automatic one, giving the user more choices and flexibility. We show by means of examples and experiments the usefulness of our approach.

Keywords: CLP, Dynamic slicing, Debugging, Assertion language.

1 Introduction

Constraint Logic Programming (CLP) is a language scheme [14] for combining two declarative paradigms: constraint solving and logic programming ([11] gives an overview of the various languages of the scheme and a variety of applicative areas in which CLP proved successful). Concurrent constraint programming (CCP) [19] (see a survey in [17]) combines concurrency primitives with the ability to deal with constraints, and hence, with partial information. The notion of concurrency is based upon the shared-variables communication model. CCP is intended for reasoning, modeling and programming concurrent agents (or processes) that interact with each other and their environment by posting and asking information in a medium, a so-called store. CCP is a very flexible model and has been applied to an increasing number of different fields such as probabilistic and stochastic, timed and mobile systems, and more recently to social networks with spatial and epistemic behaviors [17].

One crucial problem with constraint logic languages is to define appropriate debugging tools. Various techniques and several works have been defined for debugging these languages. Abstract interpretation techniques have been considered (e.g. in [4, 5, 9]) as well as (abstract) declarative debuggers following the seminal work of Shapiro [21]. However, these techniques are approximated (case of abstract interpretation) or it can be difficult to apply them when dealing with complex programs (case of declarative debugging) as the user should answer to too many questions.

In this paper we follow a technique inspired by slicing. Slicing was introduced in some pioneer works by Mark Weiser [24]. It was originally defined as a static technique, independent of any particular input of the program. Then, the technique was extended by introducing the so called dynamic program slicing [13]. This technique is useful for simplifying the debugging process, by selecting a portion of the program containing the faulty code. In the context of constraint logic languages we define a tool able to interact with the user and filter, in a given computation, the information which is relevant to a particular observation or result. In other words, the programmer could mark the information (constraints and agents or atoms) that she is interested to check in a particular computation that she suspects to be wrong. Then, a corresponding depurated partial computation is obtained automatically, where only the information relevant to the marked parts is present. Dynamic program slicing has been applied to several programming paradigms, for instance to imperative programming [13], functional programming [16], Term Rewriting [1], and functional logic programming [2]. See [12] for a survey.

In a previous paper [8] we presented the first formal framework for CCP dynamic slicing. Our aim was to help the programmer to debug her program, in cases where she could not find the bugs by using other debuggers. In this paper we investigate an extension of the framework to Constraint Logic Programs (CLP), and we try to automatize the slicing process by integrating it with a suitable assertion language. The extension to CLP is not immediate, as while for CCP programs non-deterministic choices give rise to one single computation, in CLP all computations corresponding to different non-deterministic choices can be followed and can lead to different solutions. So, some rethinking of the the framework is necessary. We show that it is possible to define a transformation from CLP programs to CCP programs, which allows to show that the set of observables of one CLP program and of the corresponding translated CCP program correspond. This result also shows that the computations in the two languages are pretty similar and the framework for CCP can be extended to deal with CLP programs. Our framework [8] consists of three main steps. First the standard operational semantics of the sliced language is extended to a "collecting semantics" that adds the needed information for the slicer. Second, we consider several analyses of the faulty situation based on error symptoms, including causality, variable dependencies, unexpected behaviors and store inconsistencies. This second step was left to the user's responsibility in our previous work [8]. The user had to examine a state of a partial computation that she recognized to be wrong. Then she had to mark some information (a subset of constraints in the last state) that she wanted to study further, removing the information in the computation not relevant to derive the marked one. Thirdly, we considered an automatic marking algorithm of the redundant items and define a trace slice. This algorithm was flexible and applicable to timed extensions of CCP [18]. Here, for CLP programs we introduce also the possibility to mark atoms, besides constraints.

We believe that the second step above, namely identifying the right state and the relevant information to be marked, can be difficult for the user and we believe that it is possible to improve automatization of this step. For this reason, in this paper we introduce a specialized assertion language which allows the user to state properties of the computations in her program. If a state in a computation does not satisfy an assertion then some "wrong" information is identified and an automatic slicing process can start. We show that assertions can be integrated in our previous semi-automatic framework [8], giving the user more choices and flexibility. The assertion language is a good companion to the already implemented tool for the slicing of CCPprograms to automatically deduce (possible) symptoms and stop the computation when need it. The framework can also be applied to timed variants of CCP.

Organization. Section 2 describes CCP and CLP and their operational semantics. In this section we also introduce a translation from CLP to CCP programs and prove a correspondence theorem between successful computations. In Section 3 we recall the slicing technique for CCP and extend it to CLP. Then, in Section 4 we present our specialized assertion language and describe its main operators and functionalities. In Section 4.2 we show some examples to illustrate the expressive power of our extension, and that it can be integrated into the former tool. Within our examples we show how to automatically debug a biochemical system specified in timed CCP and one classical search problem in CLP. Finally, Section 5 discusses some related work and concludes.

2 Constraint Logic Languages

In this section we define an operational semantics suitable for both, Constraint Logic Programming [11] and for CCP programs. We start by defining CCP programs and then we obtain CLP programs by restricting the set of operators in CCP. More precisely, we remove the synchronization operator ($\mathbf{ask}\ (c)\ \mathbf{then}\ P$) and we interpret non-determinism in a different manner. In practice, in CLP, we have to consider non-determinism of the type "don't know" [20], which means that each predicate call can be reduced by using each rule which defines such predicate. This is different from the kind of non-determinism in CCP, where the choice operator selects randomly one of the choices whose ask guard is entailed by the constraints in the current store.

Processes in CCP *interact* with each other by *telling* and *asking* constraints (pieces of information) in a common store of partial information. The type of constraints is not fixed but parametric in a constraint system (CS). The notion of CS is central to both CCP and CLP. Intuitively, a CS provides a signature from which constraints can be built from basic tokens (e.g., predicate symbols), and two basic operations: conjunction ($\sqcup$) and variable hiding ($\exists$). The CS defines also an *entailment* relation ($\models$) specifying interdependencies between constraints: $c \models d$ means that the information d can be deduced from the information c. Following [6], a cylindric algebra gives a general notion of constraint system (see the details, e.g., in [9]).

A cylindric constraint system is a structure $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \mathtt{t}, \mathtt{f}, \mathit{Var}, \exists, D\rangle$ s.t. $\langle \mathcal{C}, \leq, \sqcup, \mathtt{t}, \mathtt{f}\rangle$ is a complete algebraic lattice with $\sqcup$ the *lub* operation (representing *conjunction*). Elements in $\mathcal{C}$ are called *constraints* with typical elements c, c', d, d'..., and $\mathtt{t}$, $\mathtt{f}$ the least and the greatest elements. If $c \leq d$, we say that d entails c and we write $d \models c$. Var is a denumerable set of variables and for each $x \in \mathit{Var}$ the function $\exists x : \mathcal{C} \to \mathcal{C}$ is a cylindrification operator (representing information hiding). As usual, $\exists x.c$ binds x in c. We use $\mathit{fv}(\cdot)$ (resp. $\mathit{bv}(\cdot)$) to denote the set of free (resp. bound) variables. For each $x, y \in \mathit{Var}$, the constraint $d_{xy} \in D$ is a *diagonal element* can be thought of as the equality $x = y$, useful to define substitutions of the form $[t/x]$.

As an example, consider the finite domain constraint system (FD) [10]. This system assumes variables to range over finite domains and, in addition to equality, one may have predicates that restrict the possible values of a variable as in $x < 42$.

The language of CCP processes. In the spirit of process calculi, the language of processes in CCP is given by a small number of primitive operators or combinators. Processes are built from constraints in the underlying constraint system and the syntax:

$$P, Q ::= \mathbf{skip} \mid \mathbf{tell}(c) \mid \sum_{i \in I} \mathbf{ask}\ (c_i)\ \mathbf{then}\ P_i \mid P \parallel Q \mid (\mathbf{local}\, x)\, P \mid p(\overline{x})$$

The process $\mathbf{skip}$ represents inaction. The process $\mathbf{tell}(c)$ adds c to the current store d producing the new store $c \sqcup d$. Given a non-empty finite set of indexes I, the process $\sum_{i \in I} \mathbf{ask}\ (c_i)\ \mathbf{then}\ P_i$ non-deterministically chooses P_k for execution if the store entails c_k. The chosen alternative, if any, precludes the others. This provides a powerful synchronization mechanism based on constraint entailment. When I is a singleton, we shall omit the "$\sum$" and we simply write $\mathbf{ask}\ (c)\ \mathbf{then}\ P$.

The process $P \parallel Q$ represents the parallel (interleaved) execution of P and Q. The process $(\mathbf{local}\, x)\, P$ behaves as P and binds the variable x to be local to it. We use $\mathit{fv}(P), \mathit{bv}(P)$ to denote, respectively, the set of free and bound variables of P.

Given a process definition $p(\overline{y}) \stackrel{\Delta}{=} P$, where all free variables of P are in the set of pairwise distinct variables $\overline{y}$, the process $p(\overline{x})$ evolves into $P[\overline{x}/\overline{y}]$. A CCP program takes the form $\mathcal{D}.P$ where $\mathcal{D}$ is a set of process definitions and P is a process.

The Structural Operational Semantics (SOS) of CCP is given by the transition relation $\gamma \longrightarrow \gamma'$ satisfying the rules in Fig. 1. Here we follow the formulation in [7] where the local variables created by the program appear explicitly in the transition system and parallel composition of agents is identified to a multiset of agents. More precisely, a *configuration* γ is a triple of the form $(X; \Gamma; c)$, where c is a constraint representing the store, Γ is a multiset of processes, and X is a set of hidden (local) variables of c and Γ. The multiset $\Gamma = P_1, P_2, \ldots, P_n$ represents the process $P_1 \parallel P_2 \parallel \cdots \parallel P_n$. We shall indistinguishably use both notations to denote parallel composition. Moreover, processes are quotiented by a structural congruence relation $\cong$ satisfying: (STR1) $P \cong Q$ if they differ only by a renaming of bound variables (alpha conversion); (STR2) $P \parallel Q \cong Q \parallel P$; (STR3) $P \parallel (Q \parallel R) \cong (P \parallel Q) \parallel R$; (STR4) $P \parallel \mathbf{skip} \cong P$.

Let us briefly explain Figure 1. A tell agent $\mathbf{tell}(c)$ adds c to the current store d (Rule $\mathrm{R_{TELL}}$); the process $\sum_{i \in I} \mathbf{ask}\ (c_i)\ \mathbf{then}\ P_i$ executes P_k if its corresponding guard c_k can be entailed from the store (Rule $\mathrm{R_{SUM}}$); a local process $(\mathbf{local}\, x)\, P$ adds x to the set of hidden variable X when no clashes of variables occur (Rule $\mathrm{R_{LOC}}$). Observe

$$\frac{}{(X;\mathbf{tell}(c),\Gamma;d)\longrightarrow(X;\mathbf{skip},\Gamma;c\sqcup d)}\ \mathrm{R_{TELL}} \qquad \frac{d\models c_k \quad k\in I}{(X;\sum\limits_{i\in I}\mathbf{ask}\ (c_i)\ \mathbf{then}\ P_i,\Gamma;d)\longrightarrow(X;P_k,\Gamma;d)}\ \mathrm{R_{SUM}}$$

$$\frac{x\notin X\cup fv(d)\cup fv(\Gamma)}{(X;(\mathbf{local}\,x)\,P,\Gamma;d)\longrightarrow(X\cup\{x\};P,\Gamma;d)}\ \mathrm{R_{LOC}} \qquad \frac{p(\overline{y})\overset{\Delta}{=}P\in\mathcal{D}}{(X;p(\overline{x}),\Gamma;d)\longrightarrow(X;P[\overline{x}/\overline{y}],\Gamma;d)}\ \mathrm{R_{CALL}}$$

$$\frac{(X;\Gamma;c)\cong(X';\Gamma';c')\longrightarrow(Y';\Delta';d')\cong(Y;\Delta;d)}{(X;\Gamma;c)\longrightarrow(Y;\Delta;d)}\ \mathrm{R_{EQUIV}}$$

Fig. 1: Operational semantics for CCP calculi

that Rule $\mathrm{R_{EQUIV}}$ can be used to do alpha conversion if the premise of $\mathrm{R_{LOC}}$ cannot be satisfied; finally the call $p(\overline{x})$ executes the body of the process definition (Rule $\mathrm{R_{CALL}}$).

Definition 1 (Observables). *Let $\longrightarrow^*$ denote the reflexive and transitive closure of $\longrightarrow$. If $(X;\Gamma;d)\longrightarrow^*(X';\Gamma';d')$ and $\exists X'.d'\models c$ we write $(X;\Gamma;d)\Downarrow_c$. If $X=\emptyset$ and $d=\mathtt{t}$ we simply write $\Gamma\Downarrow_c$.*

Intuitively, if P is a process then $P\Downarrow_c$ says that P can reach a store d strong enough to entail c, *i.e.*, c is an output of P. Note that the variables in X' above are hidden from d' since the information about them is not observable.

2.1 The language of CLP

A CLP program [14] is a finite set of rules of the form

$$A\leftarrow A_1,\ldots,A_n$$

where A is an atom, and $A_1,\ldots A_n$, with $n\geq 0$, are literals, i.e. either atoms or primitive constraints. An atom has the form $p(t_1,\ldots,t_m)$, where p is a user defined predicate symbol and the t_i are terms from the constraint domain.

The top-down operational semantics is given in terms of derivations from goals [14]. A configurations takes the form $(\Gamma;c)$ where Γ (a goal) is a multiset of literals and c is a constraint (the current store). The reduction relation is defined as follows.

Definition 2 (Semantics of CLP [14]). *A configuration $\gamma=(L_1,...,L_i,...L_n;c)$ reduces to ψ, notation $\gamma\longrightarrow_{CLP}\psi$, by selecting a literal L_i and then:*

1. *If L_i is a constraint d and $d\sqcup c\neq\mathtt{f}$, then $\gamma\longrightarrow_{CLP}(L_1,...,L_n;c\sqcup d)$.*
2. *If L_i is a constraint d and $d\sqcup c=\mathtt{f}$ (i.e., the conjunction of c and d is inconsistent), then $\gamma\longrightarrow_{CLP}(\Box;\mathtt{f})$ where $\Box$ represents the empty multiset of literals.*
3. *If L_i is a user defined predicate $A(t_1,...,t_k)$, then $\gamma\longrightarrow_{CLP}(L_1,...,\Delta,...,L_n;c)$ where one of the definitions for A, $A(s_1,...,s_k)\leftarrow A_1,\ldots,A_n$, is selected and $\Delta=A_1,\ldots,A_n,s_1=t_1,...,s_k=t_k$.*

A computation from a goal G is a (possibly infinite) sequence $\gamma_1=(G;\mathtt{t})\longrightarrow_{CLP}\gamma_2\longrightarrow_{CLP}\cdots$. We say that a computation finishes if the last configuration γ_n cannot be reduced, i.e., $\gamma_n=(\Box;c)$. If $c=\mathtt{f}$ then the derivation fails otherwise it succeeds.

We shall use $\longrightarrow^*_{CLP}$ to denote the reflexive and transitive closure of $\longrightarrow_{CLP}$. Following Definition 1 for CCP, given a goal with free variables $\overline{x} = var(G)$, we shall also use the notation $G \Downarrow_c$ to denote that there is a successful computation $(G; \mathtt{t}) \longrightarrow^*_{CLP} (\Box; d)$ s.t. $\exists \overline{x}.d \models c$. We note that the free variables of a goal are progressively "instantiated" during computations by adding new constraints. Finally, the answers of a goal G, notation $G \Downarrow$ is the set $\{\exists_{var(c)\setminus var(G)}(c) \mid (G; \mathtt{t}) \longrightarrow^*_{CLP} (\Box; c), c \neq \mathtt{f}\}$.

The CCP model traces its origins back to the ideas of computing with constraints, Concurrent Logic Programming and Constraint Logic Programming (CLP) [19]. Hence, CCP can simulate computations of such models. In Definition 3 below, we give a CCP interpretation to single computations in CLP programs. We emphasize that one execution of a CCP program will give rise to a single computation (due to the kind of non-determinism in CCP) while the CLP abstract computation model characterizes the set of all possible successful derivations and corresponding answers. In other terms, for a given initial goal G, the CLP model defines the full set of answer constraints for G, while the CCP translation will compute only one of them, as only one possible derivation will be followed.

Definition 3 (Translation). *Let $\mathcal{C}$ be a constraint system and $\mathcal{H}$ be a CLP program consisting of a set of clauses and G be a goal. We define the set of CCP process definitions $[\![\mathcal{H}]\!] = \mathcal{D}$ as follows. For each user defined predicate symbol p of arity j and $1..m$ defined clauses of the form $p(t^i_1, ..., t^i_j) \leftarrow A^i_1, \ldots, A^i_{n_i}$, we add to $\mathcal{D}$ the following process definition*

$$\begin{array}{rl} A(x_1, ..., x_j) \overset{\Delta}{=} & \mathbf{ask}\ (\exists \overline{y_1}\ \bigsqcup E_1)\ \mathbf{then}\ (\mathbf{local}\ \overline{z_1})\ \prod D_1 \ \parallel [\![A^1_1]\!] \parallel \cdots \parallel [\![A^1_{n_1}]\!] + ... + \\ & \mathbf{ask}\ (\exists \overline{y_m} \bigsqcup E_m)\ \mathbf{then}\ (\mathbf{local}\, \overline{z_m})\ \prod D_m \parallel [\![A^m_1]\!] \parallel \cdots \parallel [\![A^m_{n_m}]\!] \end{array}$$

where $\overline{y_i} = var(t^i_1, ..., t^i_j)$, $\overline{z_i} = \overline{y_i} \cup var(A^i_1, ..., A^i_{n_i})$, D_i is the set of constraints $\{x_1 = t^i_1, ..., x_j = t^i_j\}$, $E_i = \{x = t \in D_i \mid var(t) \neq \emptyset\}$, $\prod D_i$ means $\mathbf{tell}(x_1 = t^i_1) \parallel \cdots \parallel \mathbf{tell}(x_j = t^i_j)$ and literals are translated as $[\![A(\overline{t})]\!] = A(\overline{t})$ (case of atoms) and $[\![c]\!] = \mathbf{tell}(c)$ (case of constraints). Moreover, we translate the goal $[\![l_1, ..., l_n]\!]$ as the process $([\![l_1]\!] \parallel \cdots \parallel [\![l_n]\!])$.

We note that the head $p(\overline{x})$ of a process definitions $p(\overline{x}) \overset{\Delta}{=} P$ in CCP can only have variables while a head of a CLP rule $A(\overline{t}) \leftarrow B$ may have arbitrary terms with (free) variables. The set of constraints D_i allows us to introduce constraints which also establish the connection between the formal and the actual parameters of the predicates. Take for instance the following CLP rules dealing with lists:

```
p([] , []) .
p([H1 | L1] , [H2 | L2]) :- c(H1,H2), p(L,M) .
```

The translation will be

$$\begin{array}{rl} p(x, y) \overset{\Delta}{=} & \mathbf{ask}\ (\mathtt{t})\ \mathbf{then}\ \mathbf{tell}(x = [])\parallel \mathbf{tell}(y = []) + \\ & \mathbf{ask}\ (\exists X \bigsqcup D)\ \mathbf{then}\ (\mathbf{local}\, X)\ (\prod D \parallel c(H1, H2) \parallel p(L1, L2) \end{array}$$

where $D = \{x = [H1|L1], y = [H2|L2]\}$ and $X = \{H1, H2, L1, L2\}$.

Theorem 1 (Adequacy). *Let $\mathcal{C}$ by a constraint system, $\mathcal{H}$ be a set of clauses and G be a goal. Then, $G \Downarrow_c$ iff $[\![G]\!] \Downarrow_c$. (Proof in Appendix A).*

3 Slicing a CCP and CLP program

Dynamic slicing is a technique that helps the user to debug her program by simplifying a partial execution trace, thus depurating it from parts which are irrelevant to find the bug. It can also help to highlight parts of the programs which have been wrongly ignored by the execution of a wrong piece of code. In [8] we defined a slicing technique for CCP programs that consisted of three main steps:

S1 *Generating a (finite) trace* of the program. For that, a *collecting semantics* is needed in order to generate the (meta) information needed for the slicer.
S2 *Marking the final store*, to choose some of the constraints that, according to the symptoms detected, should or should not be in the final store.
S3 *Computing the trace slice*, to select the processes and constraints that were relevant to produce the (marked) final store.

We shall briefly recall the step **S1** in [8] which remains the same here. Steps **S2** and **S3** need further adjustments to deal with CLP programs. In particular, we shall allow the user to select processes (literals in the CLP terminology) in order to start the debugging. Moreover, in Section 4, we provide further tools to automatize the process of highlighting the symptoms of erros.

Collecting Semantics (Step $\mathbf{S1}$*)* The slicing process requires some extra information from the execution of the processes. More precisely, (1) in each operational step $\gamma \to \gamma'$, we need to highlight the process that was reduced; and (2) the constraints accumulated in the store must reflect, exactly, the contribution of each process to the store. In order to solve (1) and (2), we introduced in [8] the collecting semantics that extracts the needed meta information for the slicer. Roughly, we identify the parallel composition $Q = P_1 \parallel \cdots \parallel P_n$ with the *sequence* $\Gamma_Q = P_1 : i_1, \cdots, P_n : i_n$ where $i_j \in \mathbb{N}$ is a unique identifier for P_j. The use of indexes allow us to distinguish, e.g., the three different occurrences of P in "$\Gamma_1, P : i, \Gamma_2, P : j, (\mathbf{ask}\ (c)\ \mathbf{then}\ P) : k$". The collecting semantics uses transitions with labels of the form $\xrightarrow{[i]_k}$ where i is the identifier of the reduced process and k can be either $\perp$ (undefined) or a natural number indicating the branch chosen in a non-deterministic choice (Rule $\mathrm{R}'_{\mathrm{SUM}}$). This allows us to identify, unequivocally, the selected alternative in an execution. Finally, the store in the collecting semantics is not a constraint (as in Fig. 1) but a set of (atomic) constraints where $\{d_1, \cdots, d_n\}$ represents the store $d_1 \sqcup \cdots \sqcup d_n$ in the operational semantics. For that, the rule of $\mathbf{tell}(c)$ first decomposes c in its atomic components before add them to the store.

Marking the Store (Step $\mathbf{S2}$*).* In [8] we identified several alternatives for marking the final store in order to indicate the symptoms that are relevant to the slice that the programmer wants to recompute. Let us suppose that the final configuration in a partial computation is $(X; \Gamma; S)$. The symptoms that something is wrong may be (and not limited to) the following:

1. *Causality:* the user identifies, according to her knowledge, a subset $S' \subseteq S$ that needs to be explained (i.e., we need to identify the processes that produced S').

Input: - a trace $\gamma_0 \xrightarrow{[i_1]_{k_1}} \cdots \xrightarrow{[i_n]_{k_n}} \gamma_n$ where $\gamma_i = (X_i; \mathit{\Gamma}_i; S_i)$
- a marking $(S_{sliced}, \mathit{\Gamma}_{sliced})$ s.t. $S_{sliced} \subseteq S_n$ and $\mathit{\Gamma}_{sliced} \subseteq \mathit{\Gamma}_n$

Output: a sliced trace $\gamma'_0 \longrightarrow \cdots \longrightarrow \gamma'_n$

1 **begin**
2   **let** $\theta = \emptyset$ **in**
3   $\gamma'_n \leftarrow (X_n \cap vars(S_{sliced}); \mathit{\Gamma}_{sliced}; S_{sliced});$
4   **for** l= $n-1$ *to* 0 **do**
5     $\mathbf{let} \langle \theta', c\rangle = sliceProcess(\gamma_l, \gamma_{l+1}, i_{l+1}, k_{l+1}, \theta, S) \circ \theta$ **in**
6     $S_{sliced} \leftarrow S_{sliced} \cup S_{minimal}(S_l, c)$
7     $\theta \leftarrow \theta' \circ \theta$
8     $\gamma'_l \leftarrow (X_l \cap vars(S_{sliced}, \mathit{\Gamma}_{sliced})\ ;\ \mathit{\Gamma}_l\theta\ ;\ S_l \cap S_{sliced})$
9   **end**
10 **end**

Algorithm 1: Trace Slicer. $S_{minimal}(S, c) = \emptyset$ if $c = \mathtt{t}$; otherwise, $S_{minimal}(S, c) = \bigcup\{S' \subseteq S \mid \bigsqcup S' \models c$ and S' is set minimal$\}$.

2. *Variable Dependencies:* The user may identify a set of relevant variables $V \subseteq fv(S)$ and then, we mark $S_{sliced} = \{c \in S \mid vars(c) \cap V \neq \emptyset\}$.
3. *Unexpected behaviors*: there is a constraint c entailed from the final store that is not expected from the intended behavior of the program. Then, one would be interested in the following marking $S_{sliced} = \bigcup\{S' \subseteq S \mid \bigsqcup S' \models c$ and S' is set minimal$\}$, where "S' is set minimal" means that for any $S'' \subset S'$, $S'' \not\models c$.
4. *Inconsistent output*: The final store should be consistent with respect to a given specification (constraint) c, i.e., S in conjunction with c must not be inconsistent. In this case, we have $S_{sliced} = \bigcup\{S' \subseteq S \mid \bigsqcup S' \sqcup c \models \mathtt{f}$ and S' is set minimal$\}$.

For the analysis of CLP programs, it is important also to mark literals (i.e., calls to procedures in CCP). In particular, the programmer may find that a particular goal $p(x)$ is not correct if x does not satisfy a constraint (e.g., $x > 6$). Hence, we shall consider also markings on the set of processes, i.e., the marking can be also a subset $\mathit{\Gamma}_{sliced} \subseteq \mathit{\Gamma}$. More conveniently, we shall allow markings of the form

$$\mathit{\Gamma}_{sliced} = \{p(t_1, ..., t_n) \in \mathit{\Gamma} \mid \bigsqcup S \models F\}$$

where $p(\bar{t})$ is marked if its parameters satisfy a condition F (see Def. 5 in Sec. 4).

Trace Slice (Step $\mathbf{S3}$*)* Starting from the the pair $\gamma_{sliced} = (S_{sliced}, \mathit{\Gamma}_{sliced})$ denoting the user's marking, we define a backward slicing step. Roughly, this step allows us to eliminate from the execution trace all the information not related to γ_{sliced}. For that, the fresh constant symbol $\bullet$ is used to denote an "irrelevant" constraint or process. Then, for instance, "$c \sqcup \bullet$" results from a constraint $c \sqcup d$ where d is irrelevant. Similarly in processes as, e.g., $\mathbf{ask}\ (c)\ \mathbf{then}\ (P \parallel \bullet) + \bullet$. A replacement is either a pair of the shape $[T/i]$ or $[T/c]$. In the first (resp. second) case, the process with identifier i (resp. constraint c) is replaced with T. We shall use θ to denote a set of replacements and we call these sets as "replacing substitutions". The composition of replacing substitutions θ_1 and θ_2 is given by the set union of θ_1 and θ_2, and is denoted as $\theta_1 \circ \theta_2$.

Alg. 1 extends the one in [8] to deal with the marking on processes ($\mathit{\Gamma}_{sliced}$). The last configuration (γ'_n in line 3) means that we only observe the local variables of interest, i.e., those in $vars(S_{sliced}, \mathit{\Gamma}_{sliced})$ as well as the relevant processes ($\mathit{\Gamma}_{sliced}$) and constraints (S_{sliced}). The algorithm backwardly computes the slicing by accumulating replacing pairs in θ (line 7). The new replacing substitutions are computed by the function $sliceProcess$ that returns both, a replacement substitution and a constraint needed in the case of ask agents as explained below. Definition of $sliceProcess$ is the same as in [8] and we have added it in Appendix **??**. Let us give some intuitions on how it works. Suppose that $\gamma \xrightarrow{[i]_k} \psi$. We consider each kind of process. For instance, assume a $\mathrm{R}'_{\mathrm{TELL}}$ transition $\gamma = (X_\gamma; \Gamma_1, \mathbf{tell}(c)\!:\!i, \Gamma_2; S_\gamma) \xrightarrow{[i]} (X_\psi; \Gamma_1, \Gamma_2; S_\psi) = \psi$. We note that $X_\gamma \subseteq X_\psi$ and $S_\gamma \subseteq S_\psi$ and the contribution of $\mathbf{tell}(c)$ to the store is $S_c = S_\psi \setminus S_\gamma$. We replace the constraint c with its sliced version c' where any atom $c_a \in S_c$ not in the relevant set of constraints S_{sliced} is replaced by $\bullet$. By joining together the resulting atoms, and existentially quantifying the variables in $X_\psi \setminus X_\gamma$ (if any), we obtain the sliced constraint c'. In order to further simplify the trace, if c' is $\bullet$ or $\exists\overline{x}.\bullet$ then we substitute $\mathbf{tell}(c)$ with $\bullet$. In a non-deterministic choice, all the precluded choices are discarded ("$+\bullet$"). Moreover, if the chosen alternative Q_k does not contribute to the final store (i.e., $\Gamma_Q\theta = \bullet$), then the whole process $\sum \mathbf{ask}\ (c_l)\ \mathbf{then}\ P_l$ becomes $\bullet$. If this is not the case, besides the needed substitution replacement, we also return the constraint c_k (the entailed guard of the ask agent). Note that in line 6 of Algorithm 1, we add to S_{sliced} the minimal set of constraints that "explains" the entailed guard c_k. This allows us to highlight also the processes that added the needed information to entail such constraint.

Example 1. Consider the following (wrong) CLP program:

```
length([],0).
length([A | L],M) :- M = N, length(L, N).
```

The translation to CCP is similar to the one we gave in Section 2.1. An excerpt of a possible trace for the execution of the goal `length([10;20], Ans).` is

```
[0 ; length([10;20],Ans) ; t] -->
[0 ; ask() ... + ask() ... ; t] ->
[0 ; local ... ; t] ->
[H1 L1 N1 M1 ; [10;20]= [H1|L1] || Ans=N1 || N1=M1 || length(L1, M1) ; t] ->
...
[... H2 L2 N2 M2 ; [20]=[H2 | L2] || M1=N2 || N2=M2 || length(L2, M2) ; [10;20]= [H1|L1], Ans=N1, N1=M1] ->
[... H2 L2 N2 M2 ; M1=N2 || N2=M2 || length(L2, M2) ; [10;20]= [H1|L1], Ans=N1, N1=M1, [20]=[H2 | L2]] ->
...
[... H2 L2 N2 M2 ; M2=0 ; [10;20]= [H1|L1], Ans=N1, N1=M1, [20]=[H2 | L2], M1=N2, N2=M2, L2=[]] ->
[... H2 L2 N2 M2 ; [10;20]= [H1|L1], Ans=N1, N1=M1, [20]=[H2 | L2], M1=N2, N2=M2, L2=[], M2=0 ]
```

In the last configuration, we can mark only the equalities dealing with numerical expressions (i.e., `Ans=N1,N1=M1,M1=N2,N2=M2,M2=0`) and the resulting trace will abstract away from all the constraints and processes dealing with equalities on lists:

```
[0 ; length([10;20],Ans) ; t] -->
[0 ; * + ask() ... ; t] ->
[0 ; local ... ; t] ->
[N1 M1 ; * || Ans=N1 || N1=M1 || length(L1, M1) ; t] ->
[N1 M1 ; Ans=N1 || N1=M1 || length(L1, M1) ; ] ->
[N1 M1 ; N1=M1 || length(L1, M1) ; *, Ans=N1] ->
[N1 M1 ; length(L1, M1) ; *, Ans=N1, N1=M1] ->
...
```

The forth line should be useful to discover that `Ans` cannot be equal to `M1` (the parameter used in the second invocation to `length`).

4 An assertion language for logic programs

The declarative flavor of programming with constraints in CCP and CLP allows the user to reason about (partial) invariants that must hold during the execution of her programs. In this section we give a simple yet powerful language of assertion to state such invariants. Then, we give a step further in automatizing the process of debugging.

Definition 4 (Assertion Language). *Assertions are built from the following syntax.*
$F ::= \mathit{pos}(c) \mid \mathit{neg}(c) \mid \mathit{cons}(c) \mid \mathit{icons}(c) \mid F \oplus F \mid p(\overline{x})[F] \mid p(\overline{x})\langle F \rangle$
where c is a constraint ($c \in \mathcal{C}$), $p(\cdot)$ is a process name and $\oplus \in \{\wedge, \vee, \rightarrow\}$.

The first four constructs deal with partial assertions about the current store. These constructs check, respectively, whether the constraint c: (1) is entailed, (2) is not entailed, (3) is consistent wrt the current store or (4) leads to an inconsistency when added to the current store. Assertions of the form $F \oplus F$ have the usual meaning. The assertions $p(\overline{x})[F]$ states that all instances of $p(\overline{t})$ in the current configuration must satisfy the assertion F. The assertions $p(\overline{x})\langle F \rangle$ is similar to the previous one but it checks for the existence of an instance $p(\overline{t})$ that satisfies the the assertion F.

We shall use π to denote a trace (in the collecting semantics). Moreover, $\pi(i)$ denotes the i-th position in the sequence π. Let $\pi(i) = (X_i; \Gamma_i; S_i)$. We shall use $\mathit{store}(\pi(i))$ to denote the constraint $\exists X_i. \bigsqcup S_i$ and $\mathit{procs}(\pi(i))$ to denote the sequence of processes Γ_i. The semantics for assertions is formalized next.

Definition 5 (Semantics). *Let π be a sequence of configurations and F be an assertion. We inductively define $\pi, i \models_{\mathcal{F}} F$ (read as π satisfies the formula F at position i) as:*

- $\pi, i \models_{\mathcal{F}} \mathit{pos}(c)$ *if* $\mathit{store}(\pi(i)) \models c$.
- $\pi, i \models_{\mathcal{F}} \mathit{neg}(c)$ *if* $\mathit{store}(\pi(i)) \not\models c$.
- $\pi, i \models_{\mathcal{F}} \mathit{cons}(c)$ *if* $\mathit{store}(\pi(i)) \sqcup c \not\models \mathtt{f}$.
- $\pi, i \models_{\mathcal{F}} \mathit{icons}(c)$ *if* $\mathit{store}(\pi(i)) \sqcup c \models \mathtt{f}$.
- $\pi, i \models_{\mathcal{F}} F \wedge G$ *if* $\pi, i \models_{\mathcal{F}} F$ *and* $\pi, i \models_{\mathcal{F}} G$.
- $\pi, i \models_{\mathcal{F}} F \vee G$ *if* $\pi, i \models_{\mathcal{F}} F$ *and* $\pi, i \models_{\mathcal{F}} G$.
- $\pi, i \models_{\mathcal{F}} F \rightarrow G$ *if* $\pi, i \models_{\mathcal{F}} F$ *implies* $\pi, i \models_{\mathcal{F}} G$.
- $\pi, i \models_{\mathcal{F}} p(\overline{x})[F]$ *if for all* $p(\overline{t}) \in \mathit{procs}(\pi(i))$, $\pi, i \models_{\mathcal{F}} F[\overline{t}/\overline{x}]$.
- $\pi, i \models_{\mathcal{F}} p(\overline{x})\langle F \rangle$ *if there exists* $p(\overline{t}) \in \mathit{procs}(\pi(i))$, $\pi, i \models_{\mathcal{F}} F[\overline{t}/\overline{x}]$.

If it is not the case that $\pi, i \models_{\mathcal{F}} F$, then we say that F does not hold at $\pi(i)$ and we write $\pi(i) \not\models_{\mathcal{F}} F$.

The above definition is quite standard and reflects the intuitions given above. Moreover, let us define $\sim F$ as $\sim \mathtt{pos}(c) = \mathtt{neg}(c)$ (and vice-versa), $\sim \mathtt{cons}(c) = \mathtt{icons}(c)$ (and vice-versa), $\sim (F \oplus F)$ as usual and $\sim p(\overline{x})[F(\overline{x})] = p(\overline{x})\langle \sim F(\overline{x}) \rangle$ (and vice-versa). Note that, $\pi(i) \models_{\mathcal{F}} F$ iff $\pi(i) \not\models_{\mathcal{F}} \sim F$.

Example 2. Consider the current store in $\pi(1)$ is $S = x \in 0..10$. we then have:
- $\pi, 1 \models_{\mathcal{F}} \mathtt{cons}(x = 5)$, i.e., the current store is consistent wrt the specification $x = 5$.
- $\pi, 1 \not\models_{\mathcal{F}} \mathtt{icons}(x = 5)$, i.e., the store is not inconsistent wrt the specification $x = 5$.

- $\pi, 1 \not\models_{\mathcal{F}} \texttt{pos}(x = 5)$, i.e., the store is not "strong enough" in order to satisfy the specification $x = 5$.
- $\pi, 1 \models_{\mathcal{F}} \texttt{neg}(x = 5)$, i.e., store is "consistent enough" to guarantee that it is not the case that $x = 5$.

Note that $\pi, i \models_{\mathcal{F}} \texttt{pos}(c)$ implies $\pi, i \models_{\mathcal{F}} \texttt{cons}(c)$. However, the other direction is in general not true (as shown above). We note that CCP and CLP are monotonic in the sense that when the store c evolves into d, it must be the case that $d \models c$ (i.e., information is monotonically accumulated). Hence, $\pi, i \models \texttt{pos}(c)$ implies $\pi, i + j \models \texttt{pos}(c)$. Finally, if the store becomes inconsistent, $\texttt{cons}(c)$ does not hold for any c. Temporal [15] and linear [7] variants of CCP remove such restriction on monotonicity.

We note that checking assertions amounts, roughly, for testing the entailment relation in the underlying constraint system. Checking entailments is the basic operation CCP agents perform. Hence, from the implementation point of view, the verification of assertions does not introduce a significant extra computational cost.

Example 3 (Conditional assertions). Let us introduce some useful patterns for verification. - *Conditional constraints* : The assertion $\texttt{pos}(c) \rightarrow F$ checks for F only if c can be deduced from the store. For instance, the assertion $\texttt{pos}(c) \rightarrow \texttt{neg}(d)$ says that d must not be deduced when the store implies c. - *Conditional predicates* : Let $G = p(\overline{x})\langle \texttt{cons(t)} \rangle$. The assertion $G \rightarrow F$ states that F must be verified whenever there is a call/goal of the form $p(\overline{t})$ in the context. Moreover, $(\sim G) \rightarrow F$ verifies F when there is no calls of the form $p(\overline{t})$ in the context.

4.1 Dynamic slicing with assertions

Assertions allows the user to specify conditions that her program must satisfy during execution. If this is not the case, the program should stop and start the debugging process. In fact, the assertions may help to give a suitable marking pair $(S_{sliced}, \Gamma_{sliced})$ for the step **S2** of our algorithm as we show in the next definition.

Definition 6. *Let F be an assertion and π be a partial computation such that $\pi, n \not\models_{\mathcal{F}} F$, i.e., $\pi(n)$ fails to establish the assertion F. Let $\pi(n) = (X; \Gamma; S)$. As testing hypotheses for the symptoms of errors, we define $symp(\pi, F, n) = (S_{sl}, \Gamma_{sl})$ where*

1. *If $F = pos(c)$ then $S_{sl} = \{d \in S \mid vars(d) \cap vars(c) \neq \emptyset\}$, $\Gamma_{sl} = \emptyset$.*
2. *If $F = neg(c)$ then $S_{sl} = \bigcup\{S' \subseteq S \mid \bigsqcup S' \models c$ and S' is set minimal$\}$, $\Gamma_{sl} = \emptyset$*
3. *If $F = cons(c)$ then $S_{sl} = \bigcup\{S' \subseteq S \mid \bigsqcup S' \sqcup c \models \texttt{f}$ and S' is set minimal$\}$, $\Gamma_{sl} = \emptyset$.*
4. *If $F = icons(c)$ $S_{sl} = \{d \in S \mid vars(d) \cap vars(c) \neq \emptyset\}$ and $\Gamma_{sl} = \emptyset$.*
5. *If $F = F_1 \wedge F_2$ then $symp(\pi, F_1, n) \cup symp(\pi, F_2, n)$.*
6. *If $F = F_1 \vee F_2$ then $symp(\pi, F_1, n) \cap symp(\pi, F_2, n)$.*
7. *If $F = F_1 \rightarrow F_2$ then $symp(\pi, \sim F_1, n) \cup symp(\pi, F_2, n)$.*
8. *If $F = p(\overline{x})[F_1]$ then $S_{sl} = \emptyset$ and $\Gamma_{sl} = \{p(\overline{t}) \in \Gamma \mid \pi, n \not\models_{\mathcal{F}} F_1[\overline{t}/\overline{x}]\}$.*
9. *If $F = p(\overline{x})\langle F_1 \rangle$ then $S_{sl} = \{d \in S \mid vars(d) \cap vars(F_1) \neq \emptyset\}$, $\Gamma_{sl} = \{p(\overline{t}) \in \Gamma\}$*

Let us give some intuitions about the above definition. Consider a (partial) computation π of length n where $\pi(n) \not\models_{\mathcal{F}} F$. In the case (1) above, c must be entailed but the current store is not strong enough to do it. A good guess is to start examining the processes that added constraints using the same variables as in c. It may be the case that such processes should have added more information to entail c as expected in the specification F. Similarly for the case (4): c in conjunction with the current store should be inconsistent but it is not. Then, more information on the common variables should have been added. In the case (2), c should not be entailed but the store indeed entails c. In this case, we mark the set of constraints that entails c. The case (3) is similar. In cases (5) to (7) we use $\cup$ and $\cap$ respectively for point-wise union and intersection in the pair (S_{sl}, Γ_{sl}). This cases are self-explanatory (e.g., if $F_1 \wedge F_2$ fails, we collect the failure symptoms of either F_1 or F_2). In (8), we mark all the calls that do not satisfy the expected assertion $F(\overline{x})$. In (9), if F fails, it means that either (a) there are no calls of the shape $p(\overline{t})$ in the context or (b) none of the calls $p(\overline{t})$ satisfy F_1. For (a), similar to the case (1), a good guess is to examine the processes that added constraints with common variables to F_1 and see which one should have added more information to entail F_1. As for (b), we also select all the calls of the form $p(\overline{t})$ from the context. The reader may compare these definitions with the symptoms we proposed in Step **S2** in Section 3.

Classification of Assertions. As we explained in Section 2.1, computations in CLP can succeed or fail and the answers to a goal is the set of constraints obtained from successful computations. Hence, according to the kind of assertion, it is important to determine when the assertions in Definition 5 must stop or not the computation to start the debugging process. For that, we introduce the following classification:
- post-conditions, `post`(F) assertions : assertions that are meant to be verified only when an answer is found. This kind of assertions are used to test the "quality" of the answers wrt the specification. In this case, the slicing process begins only when an answer is computed and it does not satisfy one of the assertions. Note that assertions of the form $p(\overline{x})[F(\overline{x})]$ and $p(\overline{x})\langle F(\overline{x})\rangle$ are irrelevant as post-conditions since the set of goals in an answer must be empty.
- path invariants, `inv`(F) assertions: assertions that are meant to hold along the whole computation. Then, not satisfying an invariant must be understood as a symptom of an error and the computation must stop. We note that due to monotonicity, only assertions of the form `neg`(c) and `cons`(c) can be used to stop the computation (note that if the current configuration fails to satisfy `neg`(c), then any successor state will also fail to satisfy that assertion). Constraints of the form `pos`(c), `icons`(c) can be only checked when the answer is found since, not satisfying those conditions in the partial computation, does not imply that the final state will not satisfy them.

4.2 Experiments

We conclude this section with a series of examples showing the use of assertions. Examples 4 and 5 deal with CLP programs while Examples 6 and 7 with CCP programs.

Example 4. The debugger can automatically start and produce the same marking in Example 1 with the following (invariant) assertion:

```
length([A | L],M) :- M = N, length(L, N), inv(pos(M>0)).
```

Example 5. Consider the following CLP program (written in GNU-Prolog with integer finite domains) for solving the well known problem of posing N queens on a $N \times N$ chessboard in such a way that they do not attach to each other.

```
queens(N, Queens) :- length(Queens, N), fd_domain(Queens,1,N),
                     constrain(Queens), fd_labeling(Queens,[]).
constrain(Queens) :-fd_all_different(Queens), diagonal(Queens).
diagonal([]).
diagonal([Q|Queens]):-secure(Q, 1, Queens), diagonal(Queens).
secure(_,_,[]).
secure(X,D,[Q|Queens]) :- doesnotattack(X,Q,D),D1 is D+1, secure(X,D1,Queens).
doesnotattack(X,Y,D) :- X + D #\= Y,Y + X #\= D.
```

The program contains one mistake, which causes the introduction of a few additional and not correct solutions, e.g., `[1,5,4,3,2]` for the goal `queens(5,X).` The user now has two possible strategies: either she lets the interpreter to compute the solutions, one by one and then, when she sees a wrong solution she uses the slicer for marking manually the final store to get the sliced computation; or she can define an assertion to be verified. For instance, she can introduce the following a post-condition assertion:

```
secure(X,D,[Q|Queens]) :- doesnotattack(X,Q,D),D1 is D+1, secure(X,D1,Queens),
                          post(cons(Q #\= X+1)).
```

Now the slicer stops as soon as the constraint `X #\= Q+1` becomes inconsistent with the store in a successful computation (e.g., the assertion fails on the –partial– assignment “5,4”) and an automatic slicing of the successful computation is performed.

Example 6. In [8] we presented a compelling example of slicing for a timed CCP program modeling the synchronization of events in musical rhythmic patterns. As shown in Example 2 at `http://subsell.logic.at/slicer/`, the slicer for CCP was able to sufficiently abstract away from irrelevant processes and constraints to highlight the problem in a faulty program. However, the process of stopping the computation to start the debugging was left to the user. The property that failed in the program can be naturally expresses as an assertion. Namely, in the whole computation, if the constraint `beat` is present (representing a sound in the musical rhythm), the constraint `stop` cannot be present (representing the end of the rhythm). This can be written as the conditional assertion $\mathtt{pos}(\mathtt{beat}) \rightarrow \mathtt{neg}(\mathtt{stop})$. Following Definition 6, the constraints marked in the wrong computation are the same we considered in [8], thus automatizing completely the process of identifying the wrong computation.

Example 7. Example 3 in the URL above illustrates the use of timed CCP for the specification of biochemical systems (we invite the reader to compare in the website the sliced and non-sliced traces). Roughly, in that model, constraints of the form `Mdm2` (resp. `Mdm2A`) state that the protein $Mdm2$ is present (resp. absence). The model includes activation (and inhibition) biological rules modeled as processes (omitting some details) of the form $\mathbf{ask}\ (\mathtt{Mdm2A})\ \mathbf{then\ next\ tell}(\mathtt{Mdm2})$ modeling that “if $Mdm2$ is absent now, then it must be present in the next time-unit”. The interaction of many of these rules makes trickier the model since rules may “compete” for resources and then, we can wrongly observe at the same time-unit that $Mdm2$ is both present and absence. An assertion of the form $(\mathtt{pos}(\mathtt{Mdm2A}) \rightarrow \mathtt{neg}(\mathtt{Mdm2})) \wedge (\mathtt{pos}(\mathtt{Mdm2}) \rightarrow \mathtt{neg}(\mathtt{Mdm2A}))$ will automatically stop the computation and produce the same marking we used to depurate the program in the website.

5 Related work and conclusions

Related work Assertions for automatizing a slicing process have been previously introduced in [3] for the functional logic language Maude. The language they consider as well as the type of assertions are completely different from ours. They do not have constraints, and deal with functional and equational computations. Another previous work [22] introduced static and dynamic slicing for CLP programs. However, [22] essentially aims to identify the parts of a goal which do not share variables, so that to slice them apart. Our approach consider more situations, not only variable dependencies, but also other kinds of error symptoms. Moreover we have assertions, and hence an automatic slicing mechanism not considered in [22]. The well known debugging box model of Prolog [23] introduces a tool for observing the evolution of atoms during their reduction in the search tree. We believe that our methodology might be integrated with the box model and may extend some of its features. For instance, the box model makes basic simplifications by asking the user to specify which predicates she wants to observe. In our case one entire computational path is simplified automatically by considering the marked information and identifying the constraints and the atoms which are relevant for such information.

Conclusions and future work In this paper we have first extended a previous framework for dynamic slicing of (timed) CCP programs to the case of CLP programs. We considered a slightly different marking mechanism, extended to atoms besides constraints. Don't know non-determinism in CLP requires a different identification of the computations of interest for debugging wrt CCP. We consider different modalities specified by the user for selecting successful computations rather than all possible partial computations. As another contribution of this paper, in order to automatize the slicing process, we have introduced an assertion language. This language is rather flexible and allows to specify different types of assertions which then implies them to be applied to successful computations or to all possible partial computations. There are several different possible properties which can be specified, such as consistency with a given constraint of the constraint store, the fact that a constraint should be satisfied or not satisfied by the constraint store, a pattern to select the states on which to apply the assertions, etc. The user can specify assertions which are checked automatically and when assertions are not satisfied by a state of a selected computation then an automatic slicing of such computation can start. We implemented a prototype of the slicer in Maude and showed its use in debugging several programs including a specification of a biochemical system in CCP and a classical search problem in CLP. We are currently extending the tool to deal with CLP don't know non-determinism. We plan to add more advanced graphical tools to our prototype, as well as to study the integration of our framework with other debugging techniques, such as the box model and declarative or approximated debuggers. We also want to investigate the relation of our technique with dynamic testing (e.g. concolic techniques) and extend the assertion language with temporal operators, e.g. the past operator ($\ominus$), to better deal with temporal CCP and for expressing the relation between two consecutive temporal units.

References

1. M. Alpuente, D. Ballis, J. Espert, and D. Romero. Backward trace slicing for rewriting logic theories. In *Proc. of CADE'11*, pages 34–48, Berlin, Heidelberg, 2011. Springer-Verlag.
2. M. Alpuente, D. Ballis, F. Frechina, and D. Romero. Using conditional trace slicing for improving maude programs. *Sci. Comput. Program.*, 80:385–415, 2014.
3. M. Alpuente, D. Ballis, F. Frechina, and J. Sapiña. Debugging maude programs via runtime assertion checking and trace slicing. *J. Log. Algebr. Meth. Program.*, 85:707–736, 2016.
4. M. Codish, M. Falaschi, and K. Marriott. Suspension Analyses for Concurrent Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.
5. M. Comini, L. Titolo, and A. Villanueva. Abstract Diagnosis for Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming*, 11(4-5):487–502, 2011.
6. F. S. de Boer, A. Di Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, 151(1):37–78, 1995.
7. F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: Operational and phase semantics. *Inf. Comput.*, 165(1):14–41, 2001.
8. M. Falaschi, M. Gabbrielli, C. Olarte, and C. Palamidessi. Slicing concurrent constraint programs. In M. Hermenegildo and P. López-García, editors, *Proc.of LOPSTR 2016*, volume 10184 of *Lecture Notes in Computer Science*, pages 76–93. Springer, 2016.
9. M. Falaschi, C. Olarte, and C. Palamidessi. Abstract interpretation of temporal concurrent constraint programs. *TPLP*, 15(3):312–357, 2015.
10. P. Van Hentenryck, V. A. Saraswat, and Y. Deville. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming*, 37(1-3):139–164, 1998.
11. J. Jaffar and M. Maher. Constraint logic programming: a survey. *The Journal of Logic Programming*, 19-20(Supplement 1):503–581, 1994.
12. S. Josep. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, June 2012.
13. B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.
14. J. Jaff.ichael J. Maher, K. Marriott, and P. J. Stuckey. The semantics of constraint logic programs. *J. Log. Program.*, 37(1-3):1–46, 1998.
15. M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nord. J. Comput.*, 9(1):145–188, 2002.
16. C. Ochoa, J. Silva, and G. Vidal. Dynamic slicing of lazy functional programs based on redex trails. *Higher Order Symbol. Comput.*, 21(1-2):147–192, June 2008.
17. C. Olarte, C. Rueda, and F. D. Valencia. Models and emerging trends of concurrent constraint programming. *Constraints*, 18(4):535–578, 2013.
18. V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *J. Symb. Comput.*, 22(5/6):475–520, 1996.
19. V. A. Saraswat, M. C. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In D. S. Wise, editor, *POPL*, pages 333–352. ACM Press, 1991.
20. E. Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, 1989.
21. E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, 1983.
22. G. Szilágyi, T. Gyimóthy, and J. Maluszyński. Static and dynamic slicing of constraint logic programs. *Automated Software Engg.*, 9(1):41–65, 2002.
23. C. S. Mellish W. F. Clocksin. *Programming in Prolog*. Springer Verlag, 1981.
24. M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4):352–357, 1984.

A Proof of Adequacy

Theorem 1 (Adequacy) Let $\mathcal{C}$ by a constraint system, $\mathcal{H}$ be a set of clauses and G be a goal. Then, $G \Downarrow_c$ iff $[\![G]\!] \Downarrow_c$.

Proof. ($\Rightarrow$) The proof proceeds by induction on the derivation $\longrightarrow_{CLP}$. Assume that $(G; c) \longrightarrow_{CLP} (G'; c')$. If such reduction corresponds to adding a constraint c (cases (1) and (2) in Definition 2), then it is easy to see that the process $[\![G]\!]$ can also execute the corresponding $\mathbf{tell}(c)$ process and the result holds. Reductions of constraints representing equality on terms ((3) in Definition 2) are matched by introducing the corresponding constraint in D (Definition 3). Finally, if the reduction $(G; c) \longrightarrow_{CLP} (G'; c')$ is due to the application of (3) in Definition 2, clearly we can apply the rules $\mathrm{R_{CALL}}$ and then $\mathrm{R_{ASK}}$ to obtain the needed result. The ($\Leftarrow$) follows from similar arguments.

Marking algorithms

Function *sliceProcess*($\gamma, \psi, i, k, \theta, S$)
  let $\gamma = (X_\gamma; \Gamma, P : i, \Gamma'; S_\gamma)$ and $\psi = (X_\psi; \Gamma, \Gamma_Q, \Gamma'; S_\psi)$ **in**
  match P **with**
    case $\mathbf{tell}(c)$ **do**
      let $c' = sliceConstraints(X_\gamma, X_\psi, S_\gamma, S_\psi, S)$ **in**
      if $c' = \bullet$ *or* $c' = \exists\overline{x}.\bullet$ **then return** $\langle[\bullet/i], t\rangle$ **else return** $\langle[\mathbf{tell}(c')/i], t\rangle$;
    case $\sum \mathbf{ask}\ (c_l)\ \mathbf{then}\ Q_l$ **do**
      if $\Gamma_Q\theta = \bullet$ **then return** $\langle[\bullet/i], t\rangle$ **else return** $\langle[\mathbf{ask}\ (c_k)\ \mathbf{then}\ (\Gamma_Q\theta) + \bullet\ /\ i], c_k\rangle$;
    case $(\mathbf{local}\,x)\ Q$ **do**
      let $\{x'\} = X_\psi \setminus X_\gamma$ **in**
      if $\Gamma_Q[x'/x]\theta = \bullet$ **then return** $\langle[\bullet/i], t\rangle$ **else return** $\langle[(\mathbf{local}\,x')\ \Gamma_Q[x'/x]\theta/i], t\rangle$;
    case $p(\overline{y})$ **do**
      if $\Gamma_Q\theta = \bullet$ **then return** $\langle[\bullet/i], t\rangle$ **else return** $\langle\emptyset, t\rangle$;
  end
end
Function *sliceConstraints*($X_\gamma, X_\psi, S_\gamma, S_\psi, S$)
  let $S_c = S_\psi \setminus S_\gamma$ and $\theta = \emptyset$ **in**
  foreach $c_a \in S_c \setminus S$ **do** $\theta \leftarrow \theta \circ [\bullet/c_a]$;
  return $\exists_{X_\psi \setminus X_\gamma} . \bigsqcup S_c\theta$
end

Algorithm 2: Slicing Processes and Constraints

Algorithm 2, from [8], reported above, shows the procedures to mark processes and constraints during the backward slicing computation. The correctness of such algorithms can be stated as follows. The slicing procedure computes a suitable approximation of the concrete trace. Given two processes P, P', we say that P' approximates P, notation $P \preceq^\sharp P'$, if there exists a (possibly empty) replacement θ s.t. $P' = P\theta$ (i.e., P' is as P but replacing some subterms with $\bullet$). Let $\gamma = (X; \Gamma; S)$ and $\gamma' = (X'; \Gamma'; S')$ be two configurations s.t. $|\Gamma| = |\Gamma'|$. We say that γ' approximates γ, notation $\gamma \preceq^\sharp \gamma'$, if $X' \subseteq X$, $S' \subseteq S$ and $P_i \preceq^\sharp P'_i$ for all $i \in 1..|\Gamma|$.

Theorem 2. *(see [8]) Let* $\gamma_0 \xrightarrow{[i_1]_{k_1}} \cdots \xrightarrow{[i_n]_{k_n}} \gamma_n$ *be a partial computation and* $\gamma'_0 \xrightarrow{[i_1]_{k_1}} \cdots \xrightarrow{[i_n]_{k_n}} \gamma'_n$ *be the resulting sliced trace according to an arbitrary slicing criterion. Then, for all* $t \in 1..n$, $\gamma_t \preceq^\sharp \gamma'_t$. *Moreover, let* $Q = \sum \mathbf{ask}\ (c_k)\ \mathbf{then}\ P_k$

and assume that $(X_{t-1}; \mathit{\Gamma}, Q : i_t, \mathit{\Gamma}'; S_{t-1}) \xrightarrow{[i_t]_{k_t}} (X_t; \mathit{\Gamma}, P_{k_t} : j, \mathit{\Gamma}'; S_t)$ *for some* $t \in 1..n$. *Then,* $\exists X'_{t-1}(\bigsqcup S'_{t-1}) \models c_{k_t}$.