Slicing Concurrent Constraint Programs

M. Falaschi[1] M. Gabbrielli[2] C. Olarte[3] C. Palamidessi[4]

[1] Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche
Università di Siena, Italy.
moreno.falaschi@unisi.it.
[2] Dipartimento di Informatica - Scienza e Ingegneria, Università di Bologna, Italy.
gabbri@cs.unibo.it.
[3] ECT, Universidade Federal do Rio Grande do Norte, Brazil
carlos.olarte@gmail.com.
[4] INRIA and LIX, École Polytechnique, France.
catuscia@lix.polytechnique.fr.

Abstract. 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. Some previous works have developed (approximated) declarative debuggers for CCP languages. However, the task of debugging concurrent programs remains difficult. In this paper we define a dynamic slicer for CCP and we show it to be a useful companion tool for the existing debugging techniques. Our technique starts by considering a partial computation (a trace) that shows the presence of bugs. Often, the quantity of information in such a trace is overwhelming, and the user gets easily lost, since she cannot focus on the sources of the bugs. Our slicer allows for marking part of the state of the computation and assists the user to eliminate most of the redundant information in order to highlight the errors. We show that this technique can be tailored to timed variants of CCP. We also develop a prototypical implementation freely available for making experiments.

Keywords: Concurrent Constraint Programming, Program slicing, Debugging.

1 Introduction

Concurrent constraint programming (CCP) [23, 25] (see a survey in [20]) 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. Agents in CCP can be seen as both computing processes (behavioral style) and as logic formulae (declarative style). Hence CCP can exploit reasoning techniques from both process calculi and logic.

CCP is a very flexible model and then, it has been applied to an increasing number of different fields such as probabilistic and stochastic [4], timed [24, 17, 8] and mobile [21] systems. More recently, CCP languages have been proposed for the specification of spatial and epistemic behaviors as in social networks and cloud computing systems [14, 22].

One crucial problem when working with a concurrent language is being able to provide tools to debug programs. This is particularly useful for a language in which a program can generate a large number of parallel running agents. In order to tame this complexity, abstract interpretation techniques have been considered (e.g. in [6, 7, 11]) as well as (abstract) declarative debuggers following the seminal work of Shapiro [26]. However, these techniques are approximated (case of abstract interpretation) or can be of difficult application when dealing with complex programs (case of declarative debugging). It would be useful to have a semi automatic 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 outcome 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.

Slicing was introduced in some pioneer works by Mark Weiser [27]. It was originally defined as a static technique, not depending on a particular input. Then, the technique was extended by introducing the so called dynamic program slicing [15]. This technique is useful for simplifying the debugging process, by selecting a portion of the program containing the faulty code. Dynamic program slicing has been applied to several programming paradigms, for instance to imperative programming [15], functional programming [18], Term Rewriting [1], and functional logic programming [2]. The reader may refer [13] for a survey of the techniques defined in the literature on program slicing.

Dynamic program slicing for CCP allows us to propose a technique and a tool for helping the programmer to debug her program, in cases where she could not find the bugs by using other debuggers. In this paper we present the first formal framework for CCP dynamic slicing and show, by some working examples and a prototypical tool, the main features of this approach.

The dynamic slicing technique we propose follows three main steps. In the first step, we extend the standard operational semantics to a "collecting semantics" that adds the needed information for the slicer. In the second step, we propose several analyses of the faulty situation based on error symptoms, including causality, variable dependencies, unexpected behaviors and store inconsistencies. In step 3, based on the first two ones, we define a marking algorithm of the redundant items and define a trace slice. Our algorithm is flexible and it can deal with different variants of CCP. In particular, we show how to apply it to timed extensions of CCP [24].

Organization. Section 2 describes CCP and its operational semantics. In Section 3 we introduce a slicing technique for CCP and extend it for timed CCP in Section 4. We also present a working prototypical implementation of the slicer, together with some experiments. Finally, Section 5 discusses some hints for future work and concludes.

2 Concurrent Constraint Programming

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). 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 inter-dependencies between constraints: $c \models d$ means that the information d can be deduced from the information c. Such systems can be formalized as a Scott information system as in [25], as cylindric algebras [9], or they can be built upon a suitable fragment of logic *e.g.*, as in [17]. Here we follow [9], since the other approaches can be seen as an instance of this definition.

Definition 1 (Constraint System –CS–). *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 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* $d \models c$*. If* $c \leq d$ *and* $d \leq c$ *we write* $c \cong d$*.*
- 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 satisfying: (1)* $\exists x(c) \leq c$*. (2) If* $c \leq d$ *then* $\exists x(c) \leq \exists x(d)$*. (3)* $\exists x(c \sqcup \exists x(d)) \cong \exists x(c) \sqcup \exists x(d)$*. (4)* $\exists x \exists y(c) \cong \exists y \exists x(c)$*. (5) For an increasing chain* $c_1 \leq c_2 \leq c_3$*...,* $\exists x \bigsqcup_i c_i \cong \bigsqcup_i \exists x(c_i)$*.*
- For each $x, y \in \mathit{Var}$*, the constraint* $d_{xy} \in D$ *is a* diagonal element *and it satisfies: (1)* $d_{xx} \cong \mathtt{t}$*. (2) If* z *is different from* x, y *then* $d_{xy} \cong \exists z(d_{xz} \sqcup d_{zy})$*. (3) If* x *is different from* y *then* $c \leq d_{xy} \sqcup \exists x(c \sqcup d_{xy})$*.*

The cylindrification operators model a sort of existential quantification for hiding information. As usual, $\exists x.c$ binds x in c. We use $fv(c)$ (resp. $bv(c)$) to denote the set of free (resp. bound) variables in c. The diagonal element d_{xy} can be thought of as the equality $x = y$, useful to define substitutions of the form $[t/x]$ (see the details, e.g., in [11]).

As an example, consider the finite domain constraint system (FD) [12]. 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 $x < 42$.

2.1 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 as described below.

Definition 2 (Syntax. Indeterminate CCP language [25]). *Processes in CCP are built from constraints in the underlying CS 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 $\sum\limits_{i \in I} \mathbf{ask}\ c_i\ \mathbf{then}\ P_i$ non-deterministically chooses P_j for execution if the store entails c_j. 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 $fv(P)$ (resp. $bv(P)$) to denote the set of free (resp. bound) variables in 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 Figure 1. Here we follow the formulation in [10] 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 the rules in Figure 1. A tell agent $\mathbf{tell}(c)$ adds c to the current store d (Rule $\mathrm{R}_{\mathrm{TELL}}$); the process $\sum\limits_{i \in I} \mathbf{ask}\ c_i\ \mathbf{then}\ P_i$ executes P_j if its corresponding guard c_j can be entailed from the store (Rule $\mathrm{R}_{\mathrm{SUM}}$); a local process $(\mathbf{local}\, x)\, P$ adds x to the set of hidden variable X when no clashes of variables occur (Rule $\mathrm{R}_{\mathrm{LOC}}$). Observe that Rule $\mathrm{R}_{\mathrm{EQUIV}}$ can be used to do alpha conversion if the premise of $\mathrm{R}_{\mathrm{LOC}}$ cannot be satisfied; the call $p(\overline{x})$ executes the body of the process definition (Rule $\mathrm{R}_{\mathrm{CALL}}$).

Definition 3 (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.

3 Slicing a CCP 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.

$$\frac{}{(X;\mathbf{tell}(c),\Gamma;d) \longrightarrow (X;\Gamma;c \sqcup d)} \text{ R}_{\text{TELL}} \qquad \frac{d \models c_j \quad j \in I}{(X;\sum\limits_{i\in I} \mathbf{ask}\ c_i\ \mathbf{then}\ P_i,\Gamma;d) \longrightarrow (X;P_j,\Gamma;d)} \text{ R}_{\text{SUM}}$$

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

$$\frac{(X;\Gamma;c) \cong (X';\Gamma';c') \longrightarrow (Y';\Delta';d') \cong (Y;\Delta;d)}{(X;\Gamma;c) \longrightarrow (Y;\Delta;d)} \text{ R}_{\text{EQUIV}}$$

Fig. 1: Operational semantics for CCP calculi

$$\frac{\langle Y, S_c\rangle = atoms(c, fvars)}{(X;\Gamma,\mathbf{tell}(c),\Gamma';S) \xrightarrow{[i]} (X \cup Y;\Gamma,\Gamma';S \cup S_c)} \text{ R}'_{\text{TELL}}$$

$$\frac{\bigsqcup\limits_{d_k \in S} d_k \models c_j \quad j \in I}{(X;\Gamma,\sum\limits_{i\in I} \mathbf{ask}\ c_i\ \mathbf{then}\ P_i,\Gamma';S) \xrightarrow{[i]_j} (X;\Gamma,P_j,\Gamma';S)} \text{ R}'_{\text{SUM}}$$

$$\frac{x' \in Var \setminus fvars}{(X;\Gamma,(\mathbf{local}\,x)\,P,\Gamma';S) \xrightarrow{[i]} (X \cup \{x'\};\Gamma,P[x'/x],\Gamma';S)} \text{ R}'_{\text{LOC}}$$

$$\frac{p(\overline{y}) \stackrel{\Delta}{=} P \in \mathcal{D}}{(X;\Gamma,p(\overline{x}),\Gamma';S) \xrightarrow{[i]} (X;\Gamma,P[\overline{x}/\overline{y}],\Gamma';S)} \text{ R}'_{\text{CALL}}$$

Fig. 2: Collecting semantics for CCP calculi. Γ and Γ' are (possibly empty) sequences of processes and $i = |\Gamma| + 1$. $fvars = X \cup fv(S) \cup fv(\Gamma) \cup fv(\Gamma')$.

Our slicing technique consists of three main steps:

S1 *Generating a (finite) trace* of the program. For that, we propose a *collecting semantics* that generates 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.

3.1 Collecting Semantics (Step S1)

The rules of the SOS allow us to build a trace of a program. However, for the slicer, we need to extract more information from the execution of the processes. In particular, (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 propose a collecting semantics that captures the extra information needed for the slicer. The rules are in Figure 2 and explained below.

We identify the parallel composition $P_1 \parallel P_2 \parallel \cdots \parallel P_n$ with the *sequence* of processes $P_1, P_2, \cdots, P_n$. We shall use ϵ to denote an empty sequence of processes.

The context Γ, P, Γ' represents the fact that P is preceded and followed, respectively, by the (possibly empty) sequences of processes Γ and Γ'. Note that transitions are labelled with $\xrightarrow{[i]_j}$ where: $i = |\Gamma| + 1$ indicates the position of P in the sequence and j can be either $\perp$ (undefined) or a natural number indicating the branch chosen in a non-deterministic choice (Rule $\mathrm{R}'_{\mathrm{SUM}}$). For the sake of readability, we write $[i]$ instead of $[i]_\perp$.

Stores and Configurations. The solution for (2) amounts to consider the store, in a configuration, as a set of constraints and not as a constraint. Then, the store $\{c_1, \cdots, c_n\}$ represents the constraint $c_1 \sqcup \cdots \sqcup c_n$.

Consider the process $\mathbf{tell}(c)$ and let $V \subseteq Vars$. The Rule $\mathrm{R}'_{\mathrm{TELL}}$ first decomposes the constraint c in its atoms. For that, assume that the bound variables in c are all distinct and not in V (otherwise, by alpha conversion, we can find $c' \cong c$ satisfying such condition). We define $atoms(c, V) = \langle bv(c), basic(c) \rangle$ where

$$basic(c) = \begin{cases} c \text{ if } c \text{ is an atom}, \mathtt{t}, \mathtt{f} \text{ or } d_{xy} \\ basic(c') \text{ if } c = \exists x.c' \\ basic(c_1) \cup basic(c_2) \text{ if } c = c_1 \sqcup c_2 \end{cases}$$

Observe that in Rule $\mathrm{R}'_{\mathrm{TELL}}$, the parameter V of the function $atoms$ is the set of free variables occurring in the context, i.e., *fvars* in Figure 2. This is needed to perform alpha conversion of c (which is left implicit in the definition of *basic*) to satisfy the above condition on bound names.

Rule $\mathrm{R}'_{\mathrm{SUM}}$ signals the number of the branch j chosen for execution.

Rule $\mathrm{R}'_{\mathrm{LOC}}$ chooses a fresh variable x', i.e., a variable not in the set of free variables of the configuration (*fvars*). Hence, we execute the process $P[x'/x]$ and add x' to the set X of local variables. Rule $\mathrm{R}'_{\mathrm{CALL}}$ is self-explanatory.

It is worth noticing that we do not consider a rule for structural congruence in the collecting semantics. Such rule, in the system of Figure 1, played different roles. Axioms STR2 and STR3 provide agents with a structure of multiset (commutative and associative). As mentioned above, we consider in the collecting semantics sequences of processes to highlight the process that was reduced in a transition. The sequence Γ in Figure 2 can be of arbitrary length and then, any of the enabled processes in the sequence can be picked for execution. Axiom STR1 allowed us to perform alpha-conversion on processes. This is needed in $\mathrm{R}_{\mathrm{LOC}}$ to avoid clash of variables. Note that the new Rule $\mathrm{R}'_{\mathrm{LOC}}$ internalizes such procedure by picking a fresh variable x'. Finally, Axiom STR4 can be used to simplify **skip** processes that can be introduced, e.g., by a $\mathrm{R}_{\mathrm{TELL}}$ transition. Observe that the collecting semantics does not add any **skip** into the configuration (see Rule $\mathrm{R}'_{\mathrm{TELL}}$).

Example 1. Consider the following toy example. Let $\mathcal{D}$ contain the process definition $A \stackrel{\text{def}}{=} \mathbf{tell}(z > x + 4)$ and $\mathcal{D}.P$ be a program where

$P = \mathbf{tell}(y < 7) \parallel \mathbf{ask}\ x < 0\ \mathbf{then}\ A \parallel \mathbf{tell}(x = -3)$

The following is a possible trace generated by the collecting semantics.

$$\begin{array}{ll} & (\emptyset; \mathbf{tell}(y < 7), \mathbf{ask}\ x < 0\ \mathbf{then}\ A, \mathbf{tell}(x = -3); \mathtt{t}) \\ \xrightarrow{[1]} & (\emptyset; \mathbf{ask}\ x < 0\ \mathbf{then}\ A, \mathbf{tell}(x = -3); y < 7) \\ \xrightarrow{[2]} & (\emptyset; \mathbf{ask}\ x < 0\ \mathbf{then}\ A; y < 7, x = -3) \\ \xrightarrow{[1]_1} & (\emptyset; A; y < 7, x = -3) \\ \xrightarrow{[1]} & (\emptyset; \mathbf{tell}(z > x + 4); y < 7, x = -3) \\ \xrightarrow{[1]} & (\emptyset; \epsilon; y < 7, x = -3, z > x + 4) \end{array}$$

Now we introduce the notion of observables for the collecting semantics and we show that it coincides with that of Definition 4 for the operational semantics.

Definition 4 (Observables Collecting Semantics). *We write* $\gamma \xrightarrow{[i_1,\ldots,i_n]_{j_1,\ldots,j_n}} \gamma'$ *whenever* $\gamma = (X_0; \Gamma_0; S_0) \xrightarrow{[i_1]_{j_1}} \cdots \xrightarrow{[i_n]_{j_n}} (X_n; \Gamma_n; S_n) = \gamma'$*. Moreover, if* $\exists X_n. \bigsqcup_{d_i \in S_n} d_i \models c$*, then we write* $\gamma \Downarrow\!\!\downarrow_c$*. If* $X_0 = S_0 = \emptyset$*, we simply write* $\Gamma_0 \Downarrow\!\!\downarrow_c$*.*

Theorem 1 (Adequacy). *For any process* P *and constraint* c*,* $P \Downarrow_c$ *iff* $P \Downarrow\!\!\downarrow_c$

Proof. (*sketch*) ($\Rightarrow$) The proof proceeds by induction on the length of the derivation needed to perform the output c in $P \Downarrow_c$ and using the following results.

Given a set of variables V, a constraint d and a set of constraints S, let us use $\lfloor d \rfloor_V$ to denote (the resulting tuple) $atoms(d, V)$ and $\lceil S \rceil_V$ to denote the constraint $\exists V. \bigsqcup_{c_i \in S} c_i$. If $\langle Y, S \rangle = \lfloor d \rfloor_V$, from the definition of *atoms*, we have $d \cong \lceil S \rceil_Y$.

Let Γ (resp. Ψ) be a multiset (resp. sequence) of processes. Let us use $\lfloor \Gamma \rfloor$ to denote any sequence built from the processes in Γ and $\lceil \Psi \rceil$ to denote the multiset built from the elements in Ψ. Consider now the transition $\gamma = (X; \Gamma; d) \longrightarrow (X'; \Gamma'; d')$. Let $\langle Y, S \rangle = \lfloor d \rfloor_V$ where $V = X \cup fv(\Gamma) \cup fv(d)$. By choosing the same process reduced in γ, we can show that there exist i, j s.t. the collecting semantics mimics the same transition as $(X \cup Y, \lfloor \Gamma \rfloor, S) \xrightarrow{[i]_j} (X' \cup Y'; \lfloor \Gamma'' \rfloor; S')$ where $d' \cong \lceil S' \rceil_{Y'}$ and $\Gamma'' \cong \Gamma'$.

The ($\Leftarrow$) side follows from similar arguments.

3.2 Marking the Store (Step S2)

From the final store the user must indicate the symptoms that are relevant to the slice that she wants to recompute. For that, she must select a set of constraints that considers relevant to identify a bug. Normally, these are constraints at the end of a partial computation, and there are several strategies that one can follow to identify them.

Let us suppose that the final configuration in a partial computation is $(X; \Gamma; S)$. The symptoms that something is wrong in the program (in the sense that the user identifies some unexpected configuration) 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').
2. *Variable Dependencies:* The user may identify a set of variables $V \subseteq fv(S)$ whose constraints need to be explored. Then, one would be interested in marking the following set of constraints

$$S_{sliced} = \{c \in S \mid vars(c) \cap V \neq \emptyset\}$$

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 marking the following set of constraints:

$$S_{sliced} = \bigcup \{S' \subseteq S \mid \bigsqcup S' \models c \text{ and } S' \text{ is set minimal}\}$$

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, the set of constraints to be marked is:

$$S_{sliced} = \bigcup \{S' \subseteq S \mid \bigsqcup S' \sqcup c \models \mathtt{f} \text{ and } S' \text{ is set minimal}\}$$

where "S' is set minimal" means that for any $S'' \subset S'$, $S'' \sqcup c \not\models \mathtt{f}$.

We note that "set minimality" could be expensive to compute. However, we believe that in most of practical cases this should not be so heavy. In any case, we can always use supersets of the minimal ones which are easier to compute but less precise for eliminating useless information.

3.3 Trace Slice (Step S3)

Starting from the set S_{sliced} above we can define a backward slicing step. We shall identify, by means of a backward evaluation, the set of transitions (in the original computation) which are necessary for introducing the elements in S_{sliced}. By doing that, we will eliminate information not related to S_{sliced}.

Notation 1 (Sliced Terms) *We shall use the fresh constant symbol $\bullet$ 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, $\mathbf{ask}\ c\ \mathbf{then}\ (P \parallel \bullet) + \bullet$ results from a process of the form $\mathbf{ask}\ c\ \mathbf{then}\ (P \parallel Q) + \sum \mathbf{ask}\ c_i\ \mathbf{then}\ P_i$ where Q and the summands in $\sum \mathbf{ask}\ c_i\ \mathbf{then}\ P_i$ are irrelevant. We also assume that a sequence $\bullet, \ldots, \bullet$ with any number (≥ 1) of occurrences of $\bullet$ is equivalent to a single occurrence.*

We shall use θ to denote a set of replacements, i.e., a set of pairs of the shape $[\bullet/O]$ representing that the syntactic object O (that can be a constraint or a process) is replaced by $\bullet$. We will call these sets of replacements as "replacing substitutions". We assume that the replaced syntactic object O does not appear nested inside a bigger term. So, if $\theta = [\bullet/P]$, $P\theta = \bullet$ and $(\mathbf{ask}\ c\ \mathbf{then}\ P)\theta = \mathbf{ask}\ c\ \mathbf{then}\ P$. The composition of replacing substitutions θ_1 and θ_2 is given by the set union of the replacing pairs in θ_1 and θ_2, and is denoted as $\theta_1 \circ \theta_2$.

Algorithm 1 computes the slicing. The last configuration in the sliced trace is $(X_n \cap vars(S); \bullet; S)$. This means that we only observe the local variables of interest, i.e., those in $vars(S)$. Moreover, note that the processes in the last configuration were not executed and then, they are irrelevant (and abstracted with $\bullet$). Finally, the only relevant constraints are those in S.

Input: - a trace $\gamma_0 \xrightarrow{[i_1]_{j_1}} \cdots \xrightarrow{[i_n]_{j_n}} \gamma_n$ where $\gamma_i = (X_i; \Gamma_i; S_i)$
- a set $S \subseteq S_n$

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

begin
  let $\theta = \emptyset$ **in**
  $\gamma'_n \leftarrow (X_n \cap vars(S); \bullet; S)$;
  for k= $n-1$ *to 0* **do**
    $\theta \leftarrow sliceProcess(\gamma_k, \gamma_{k+1}, i_{k+1}, j_{k+1}, \theta, S) \circ \theta$;
    $\gamma'_k \leftarrow (X_k \cap vars(S)\ ;\ \Gamma_k\theta\ ;\ S_k \cap S)$
  end
end

Algorithm 1: Trace Slicer

The algorithm backwardly computes the slicing by accumulating replacing pairs in θ. The new replacing substitutions are computed by the function $sliceProcess$ in Algorithm 2. Suppose that $\gamma \xrightarrow{[i]_j} \psi$. We consider each kind of process.

Consider the $\mathrm{R}'_{\mathrm{TELL}}$ transition

$\gamma = (X_\gamma; \Gamma_1, \mathbf{tell}(c), \Gamma_2; S_\gamma) \xrightarrow{[i]} (X_\psi; \Gamma_1, \Gamma_2; S_\psi) = \psi.$

We note that $X_\gamma \subseteq X_\psi$ and $S_\gamma \subseteq S_\psi$. We replace the constraint c with its sliced version c' computed by the function $sliceConstraints$. In that function, we compute the contribution of $\mathbf{tell}(c)$ to the store, i.e., $S_c = S_\psi \setminus S_\psi$. Then, any atom c_a not in the relevant set of constraints S 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 $\exists \boldsymbol{x}.\bullet$ then we substitute $\mathbf{tell}(c)$ with $\bullet$ (thus avoiding the "irrelevant" process $\mathbf{tell}(\bullet)$).

In a non-deterministic choice, all the precluded choices are discarded ("$+\ \bullet$"). Moreover, if the chosen alternative P_j does not contribute to the final store (i.e., $P_j\theta = \bullet$), then the whole process $\sum \mathbf{ask}\ c_k\ \mathbf{then}\ P_k$ becomes $\bullet$. The cases for local processes and procedure calls can be explained similarly.

Example 2. Let c, d, e, f, g be constraints without any entailment and consider the following process:

$$\mathbf{ask}\ c\ \mathbf{then}\ \mathbf{tell}(e) \parallel \mathbf{ask}\ e\ \mathbf{then}\ (\mathbf{tell}(f) \parallel \mathbf{tell}(d)) \parallel \mathbf{tell}(c) \parallel \mathbf{ask}\ g\ \mathbf{then}\ \mathbf{skip}$$

In any execution of this process, the final store is $\{c, d, e, f\}$. If the user selects only f as slicing criterium, our implementation (see Section 4.1) returns the following output (that can be further simplified by collapsing the trailing list of "*"):

```
[0; * || ask(e, tell(f) || *) || * || * || * ; *][2] -->
[0; * || tell(f) || * || * || * || * ; *][2] -->
[0; * || * || * || * || * || * ; f,*][3] -->
[0; * || * || * || * || * || * ; f,*][1] --> stop
```

Note that only the relevant part of the process $\mathbf{ask}\ e\ \mathbf{then}\ (\mathbf{tell}(f) \parallel \mathbf{tell}(d))$ is highlighted as well as the process $\mathbf{tell}(f)$ that introduced f in the final store.

Function *sliceProcess*($\gamma, \psi, i, j, \theta$)
  let $\gamma = (X_\gamma; P_1, ..., P_i, ..., P_m; S_\gamma)$ and $\psi = (X_\psi; \Gamma_\psi; S_\psi)$ **in**
  match P_i **with**
    case $\mathbf{tell}(c)$
      let $c' = sliceConstraints(X_\gamma, X_\psi, S_\gamma, S_\psi, S)$ **in**
      if $c' = \bullet$ *or* $c' = \exists \boldsymbol{x}.\bullet$ **then return** $[\bullet/P_i]$ **else return** $[\mathbf{tell}(c')/P_i]$;
    case $\sum \mathbf{ask}\ c_k\ \mathbf{then}\ P_k$
      if $P_j\theta = \bullet$ **then**
        return $[\bullet/P_i]$
      else
        return $[\mathbf{ask}\ c_j\ \mathbf{then}\ P_j\theta + \bullet\ /\ P_i]$
      end
    case $(\mathbf{local}\, x)\, P$
      if $P\theta = \bullet$ **then return** $[\bullet/P_i]$ **else return** $[(\mathbf{local}\, x)\, P\theta/P_i]$;
    case $p(\overline{y})$ given that $p(\overline{x}) \stackrel{\text{def}}{=} A$
      if $A[\overline{y}/\overline{x}]\theta = \bullet$ **then return** $[\bullet/P_i]$ **else return** $\emptyset$;
  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

In the previous example, note that the process $P = \mathbf{ask}\ c\ \mathbf{then}\ \mathbf{tell}(e)$ is not selected in the trace since e is not part of the marked store. However, one may be interested in marking this process to discover the *causality* relation between this process and the process $Q = \mathbf{ask}\ e\ \mathbf{then}\ (\mathbf{tell}(f) \parallel \mathbf{tell}(d))$. Namely, P adds e to the store, needed in Q to produce f.

It turns out that we can easily adapt Algorithm 2 to capture such causality relations as follows. Assume that $sliceProcess$ returns both, a replacement θ and a constraint c, i.e., a tuple of the shape $\langle\theta, c\rangle$. In the case of $\sum \mathbf{ask}\ c_k\ \mathbf{then}\ P_k$, if $P_j\theta \neq \bullet$, we return the pair $\langle[\mathbf{ask}\ c_j\ \mathbf{then}\ P_j + \bullet\ /\ P_i], c_j\rangle$. In all the other cases, we return $\langle\theta, \mathtt{t}\rangle$ where θ is as in Algorithm 2. Intuitively, the second component of the tuple represents the guard that was entailed in a "relevant" application of the rule $\mathrm{R}'_{\mathrm{SUM}}$. Therefore, in Algorithm 1, besides accumulating θ, we add the returned guard to the set of relevant constraints S. This is done by replacing the line 5 in Algorithm 1 as follows:

$\mathbf{let}\langle\theta', c\rangle = sliceProcess(\gamma_k, \gamma_{k+1}, i_k, j_k, \theta)$ **in**
$\theta \leftarrow \theta' \circ \theta$
$S \leftarrow S \cup S_{minimal}(S_k, c)$

where $S_{minimal}(S, \mathtt{t}) = \emptyset$, otherwise, $S_{minimal}(S, c) = \bigcup\{S' \subseteq S \mid \bigsqcup S' \models c$ and S' is set minimal$\}$ and "S' is set minimal" means that for any $S'' \subset S'$, $S'' \not\models c$. Hence, we add to S the minimal set of constraints in S_k that "explains" the guard c.

Using this modified version of the algorithm, the first configuration (i.e., the first line of the output) would be:

```
[0 ; ask(c, tell(e)) || ask(e, tell(f) || *) || * || tell(c) || * ; *][3]
```

where the process $\mathbf{tell}(c)$ is also selected since the execution of $\mathbf{ask}\ c\ \mathbf{then}\ \mathbf{tell}(e)$ depends on this process.

4 Applications to Timed CCP

Reactive systems [3] are those that react continuously with their environment at a rate controlled by the environment. For example, a controller or a signal-processing system, receives a stimulus (input) from the environment, computes an output and then waits for the next interaction with the environment.

Timed CCP (`tcc`) [24, 17] is an extension of CCP tailoring ideas from Synchronous Languages [3]. More precisely, time in `tcc` is conceptually divided into *time intervals* (or *time-units*). In a particular time interval, a CCP process P gets an input c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. The resting point determines also a residual process Q that is then executed in the next time-unit. The resulting store d is not automatically transferred to the next time-unit. This way, outputs of two different time-units are not supposed to be related.

Definition 5 (Syntax of `tcc` [24, 17]). *The syntax of `tcc` is obtained by adding to Definition 2 the processes* $\mathbf{next}\ P \mid \mathbf{unless}\ c\ \mathbf{next}\ P \mid !P$.

The process $\mathbf{next}\ P$ delays the execution of P to the next time interval. We shall use $\mathbf{next}^n P$ to denote P preceded with n copies of "$\mathbf{next}$ " and $\mathbf{next}^0 P = P$.

The *time-out* $\mathbf{unless}\ c\ \mathbf{next}\ P$ is also a unit-delay, but P is executed in the next time-unit only if c is not entailed by the final store at the current time interval.

The replication $!P$ means $P \parallel \mathbf{next}\, P \parallel \mathbf{next}^2 P \parallel \ldots$, i.e., unboundedly many copies of P but one at a time. We note that in `tcc`, recursive calls must be guarded by a **next** operator to avoid infinite computations during a time-unit. Then, recursive definitions can be encoded via the ! operator [16].

The operational semantics of `tcc` considers *internal* and *observable* transitions. The internal transitions correspond to the operational steps that take place during a time-unit. The rules are the same as in Figure 2 plus:

$$\frac{\bigsqcup S \models c}{(X; \Gamma, \mathbf{unless}\ c\ \mathbf{next}\ P, \Gamma'; S) \longrightarrow (X; \Gamma, \Gamma'; S)}\ \mathrm{R_{Un}} \qquad \frac{}{(X; \Gamma, !P, \Gamma'; S) \longrightarrow (X; \Gamma, P, \mathbf{next}\ !P, \Gamma'; S)}\ \mathrm{R_!}$$

The $\mathbf{unless}$ process is precluded from execution if its guard can be entailed from the current store. The process $!P$ creates a copy of P in the current time-unit and it is executed in the next time-unit. The seemingly missing rule for the $\mathbf{next}$ operator is clarified below.

The *observable transition* $P \xRightarrow{(c,d)} Q$ should be read as "P on input c, reduces in one *time-unit* to Q and outputs d". The observable transitions are obtained from finite sequences of internal ones, i.e.,

$$\frac{(\emptyset; \Gamma; c) \longrightarrow^{*} (X; \Gamma'; c') \not\longrightarrow}{\Gamma \xRightarrow{(c, \exists X.c')} (\mathbf{local}\, X)\, F(\Gamma')} \ \mathrm{R_{Obs}}$$

The process $F(\Gamma')$ (the continuation of Γ') is obtained as follow:

$$F(R) = \begin{cases} \mathbf{skip} & \text{if } R = \mathbf{skip} \text{ or } R = \mathbf{ask}\ c\ \mathbf{then}\ R' \\ F(R_1) \parallel F(R_2) & \text{if } R = R_1 \parallel R_2 \\ Q & \text{if } R = \mathbf{next}\ Q \text{ or } R = \mathbf{unless}\ c\ \mathbf{next}\ Q \end{cases}$$

The function $F(R)$ (the future of R) returns the processes that must be executed in the next time-unit. More precisely, it unfolds *next* and *unless* expressions. Notice that an *ask* process reduces to $\mathbf{skip}$ if its guard was not entailed by the final store. Notice also that F is not defined for $\mathbf{tell}(c)$, $!Q$, $(\mathbf{local}\, x)\, P$ or $p(\overline{x})$ processes since all of them give rise to an internal transition. Hence these processes can only appear in the continuation if they occur within a $\mathbf{next}$ or $\mathbf{unless}$ expression.

4.1 A trace Slicer for `tcc`

From the execution point of view, only the observable transition is relevant since it describes the input-output behavior of processes. However, when a `tcc` program is debugged, we have to consider also the internal transitions. This makes the task of debugging even harder when compared to CCP.

We implemented in Maude (`http://maude.cs.illinois.edu`) a prototypical version of a slicer for `tcc` (and then for CCP) that can be found at `http://subsell.logic.at/slicer/`.

The slicing technique for the internal transition is based on the Algorithm 1 by adding the following cases to Algorithm 2:

case unless c **next** P
  return $[\bullet/P_i]$
case $!P$
  if $P\theta = \bullet\ and\ (\mathbf{next}\ !P)\theta = \bullet$ **then**
    return $[\bullet/P_i]$
  else
    return $[!(P\theta)/P_i]$
  end

Note that if an **unless** process evolves during a time-unit, then it is irrelevant. In the case of $!P$, we check whether P is relevant in the current time-unit ($P\theta$) or in the following one ($(\mathbf{next}\ !P)\theta$). If this is not the case, then $!P$ is irrelevant. Recall that **next** processes do not exhibit any transition during a time-unit and then, we do not consider this case in the extended version of Algorithm 2.

For the observable transition we proceed as follows. Consider a trace of n observable steps $\gamma_0 \Longrightarrow \cdots \Longrightarrow \gamma_n$ and a set S_{slice} of relevant constraints to be observed in the last configuration γ_n. Let θ_n be the replacement computed during the slicing process of the (internal) trace generated from γ_n. We propagate the replacements in θ_n to the configuration γ_{n-1} as follows:

1. In γ_{n-1} we set $S_{sliced} = \emptyset$. Note that the unique store of interest for the user is the one in γ_n. Recall also that the final store in `tcc` is not transferred to the next time-unit. Then, only the processes (and not the constraints) in γ_{n-1} are responsible for the final store in γ_n.
2. Let ψ be the last internal configuration in γ_{n-1}, i.e., $\gamma_{n-1} \longrightarrow^* \psi \not\longrightarrow$ and $\gamma_n = F(\psi)$. We propagate the replacements in θ_n to ψ before running the slicer on the trace starting from γ_{n-1}. For that, we compute a replacement θ' that must be applied to ψ as follows:
   – If there is a process $R = \mathbf{next}\ P$ in ψ, then θ' includes the substitution $[\mathbf{next}\ (P\theta_n)/\mathbf{next}\ P]$. For instance, if $R = \mathbf{next}\ (\mathbf{tell}(c) \parallel \mathbf{tell}(d))$ and $\mathbf{tell}(c)$ was irrelevant in γ_n (i.e., $[\bullet/\mathbf{tell}(c)] \in \theta_n$), we apply the substitution $[\mathbf{next}\ (\bullet \parallel \mathbf{tell}(d))/R]$ in ψ. The case for $\mathbf{unless}\ c\ \mathbf{next}\ P$ is similar.
   – If there is a process $R = \mathbf{ask}\ c\ \mathbf{then}\ P$ in ψ (which is irrelevant since it was not executed), we add to θ' the replacement $[\bullet/R]$.
3. Starting from $\psi\theta$, we compute the slicing on γ_{n-1} (Algorithm 1)
4. This procedure continues until the first configuration γ_0 is reached.

In the following example we show some experiments that can be found at tool's web page.

Example 3. Consider the following process definitions:

$$System \stackrel{\text{def}}{=} Beat2 \parallel Beat4$$
$$Beat2 \stackrel{\text{def}}{=} \mathbf{tell}(b2) \parallel \mathbf{next}^{\ 2}\ Beat2$$
$$Beat4 \stackrel{\text{def}}{=} \mathbf{tell}(b4) \parallel \mathbf{next}^{\ 4}\ Beat4$$

This is a simple model of a multimedia system that, every 2 (resp. 4) time-units, produce the constraint $b2$ (resp. $b4$). Then, every 4 time-units, the system produces both $b2$ and $b4$. If we compute 5 time-units and choose $S_{slice} = \{b4\}$ we obtain:

```
{1 / 5 > [System ; *] -->  [Beat4 ; *] --> [next^4(Beat4) ; *]} ==>
{2 / 5 > [next^3(Beat4) ; *]} ==>
{3 / 5 > [next^2(Beat4) ; *] } ==>
{4 / 5 > [next(Beat4) ; *]} ==>
{5 / 5 > [Beat4 ; *] --> [tell(b4) || * ; *] --> [* ; b4]}
```

Note that all the executions of $Beat2$ in time-units 1, 3 and 5 are hidden since they do not contribute to the observed output $b4$. More interestingly, the execution of $\mathbf{tell}(b4)$ in time-unit 1, as well as the recursive call of $Beat4$ ($\mathbf{next}^{\ 4}\ Beat4$) in time-unit 5, are also hidden.

Now assume that we compute an even number of time-units. Then, no constraint is produced in that time-unit and the whole execution of $System$ is hidden:

```
{1/4 > [* ; *]} ==> {2/4 > [* ; *]} ==>
{3/4 > [* ; *]} ==> {4/4 > [* ; *]}
```

As a more compelling example, consider the following process definitions:

$$
\begin{aligned}
Beat &\stackrel{\text{def}}{=} \prod_{i \in I_1} \textbf{next}\ ^{i}\textbf{tell}(\texttt{beat}) \\
Start &\stackrel{\text{def}}{=} \sum_{i \in I_2} \textbf{next}\ ^{i}(\textbf{tell}(\texttt{start})) \\
Check &\stackrel{\text{def}}{=} !\textbf{ask}\ \texttt{start}\ \textbf{then}\ \textbf{next}\ ^{12}(\textbf{tell}(\texttt{stop})) \\
System &\stackrel{\text{def}}{=} Beat \parallel Start \parallel Check
\end{aligned}
$$

where $I_1 = \{0, 3, 5, 7, 9, 11, 14, 16, 18, 20, 22\}$, $I_2 = \{0, 3, 5, 7, 9, 11\}$ and Π_i stands for parallel composition. This process represents a rhythmic pattern where groups of "2"-unit elements separate groups of "3"-unit elements, e.g., $3\ \underbrace{2\,2\,2\,2}\ 3\ \underbrace{2\,2\,2\,2\,2}$. Such pattern appears in repertoires of Central African Republic music [5] and were programmed in `tcc` in [19].

This pattern can be represented in a circle with 24 divisions, where "2" and "3"-unit elements are placed. The "3"-unit intervals are displayed in red in Figure 3. The important property is *asymmetry*, i.e., if one attempts to break the circle into two parts, it is not possible to have two equals parts. To be more precise, the `start` and `stop` constraints divide the circle in two halves (see process Start) and it is always the case that the constraint `beat` does not coincide in a time-unit with the constraint `stop`. For instance, in Figure 3 (a) (resp. (b)), the circle is divided in time-units 1 –start– to 13 –stop– (resp. 4 –start– to 16 –stop–). The signal `beat` does not coincide with a `stop`: in Figure 3 (a) (resp. (b)), the `beat` is added in time-unit 12 (resp. 15).

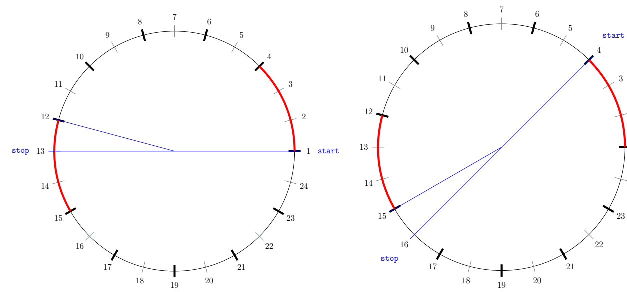
If we generate the slice for the time-unit 13 with $S_{sliced} = \{\texttt{beat}, \texttt{stop}\}$, we only observe as relevant process $Check$ (since no `beat` is produced in that time-unit) :

```
{1 / 13 > [System ; *] --> [Check ; *] --> [! ask(start, next^12(tell(stop)) ; *]
          --> [ask(start, next^12(tell(stop)) ; *] --> [next^12(tell(stop) ; *]} ==>
.... ==> ...
{11 / 13 > [next(next(tell(stop))) ; *]} ==>
{12 / 13 > [next(tell(stop)) || * ; *]} ==>
{13 / 13 > [tell(stop) ; *] --> [* ; stop][0]}
```

More interesting, assume that we wrongly write a process $Check$ that is not "well synchronized" with the process $Beat$. For instance, if $I'_2 = \{2\}$, then the `start` signal does not coincide with a `beat`. Then, in time-unit 15, we (wrongly) observe both `beat` and `stop`. The trace of that program (that can be found in tool's web page) is quite long and difficult to understand. On the contrary, the sliced one is rather simple:

```
{1 / 15 > [System ; *] --> [Beat || Check ; *] -->
          [next^14(tell(beat) || next(! ask(start, next^12(tell(stop)); *]} ==>
{2 / 15 > [next^13(tell(beat))|| ! ask(start, next^12(tell(stop))) ; *]} ==>
{3 / 15 > [next^12(tell(beat)))|| ! ask(start, next^12(tell(stop)) ; *]} ==>
{4 / 15 > [next^11(tell(beat))|| next^11(tell(stop)|| * ; *] --> stop} ==>
...
{14 / 15 > [next(tell(beat)) || next(tell(stop)) || * ; *] --> stop} ==>
{15 / 15 > [tell(beat) || tell(stop) || * ; *] --> [tell(stop) || * ; beat] -->
           [* ; beat,stop]}
```

Something interesting in this trace is that the **ask** in $Check$ is hidden from the time-unit 4 on (since it is not "needed" any more). Moreover, the only **tell**(`beat`) process (from $Beat$ definition) displayed is the one that is executed in time-unit 15 (i.e., the one



(a) (b)

Fig. 3: Pattern of "2" and "3"-unit elements (taken from [5]).

resulting from $\mathbf{next}\ ^{14}\mathbf{tell}(\mathtt{beat})$). From this trace, it is not difficult to note that the $Start$ process begins its execution on time-unit 3 (the process $\mathbf{next}\ ^{11}\mathbf{tell}(\mathtt{stop})$ first appears on time-unit 4). This can tell the user that the process $Start$ begins its execution in a wrong time-unit. In order to confirm this, the user may compute the sliced trace up to time-unit 3 with $S_{sliced} = \{\mathtt{beat}, \mathtt{start}\}$ and notice that, in that time-unit, `start` is produced but `beat` is not part of the store.

5 Conclusions and future work

In this paper we introduced the first framework for slicing concurrent constraint based languages, and showed its applicability for CCP and timed CCP. Our framework is a good basis for dealing with other variants of CCP such as linear CCP [10], spatial and epistemic CCP [14] as well as with other temporal extensions of it [8], which we are currently investigating. We implemented an initial prototype of the slicer in Maude and showed its use in debugging a program specifying a multimedia interacting system.

As future work we are considering another kind of symptom that we can identify in a bugged program. This happens when a constraint c should be part of the final store S but S does not entail it. Note that this case does not fit into the schemes proposed in Section 3.2. To cover this analysis, we are currently working on a new feature of our tool that marks the processes that may produce c or a constraint entailing c and were not executed.

Acknowledgements. We thank the anonymous reviewers for their detailed comments and suggestions which helped us to improve our paper. The work of Olarte was funded by CNPq (Brazil).

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. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
4. L. Bortolussi and A. Policriti. Modeling biological systems in stochastic concurrent constraint programming. *Constraints*, 13(1-2):66–90, 2008.
5. M. Chemillier. *Les Mathématiques Naturelles*. Odile Jacob, 2007.
6. 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.
7. 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.
8. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Inf. Comput.*, 161(1):45–83, 2000.
9. 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.
10. F. Fages, P. Ruet, and S. Soliman. Linear concurrent constraint programming: Operational and phase semantics. *Inf. Comput.*, 165(1):14–41, 2001.
11. M. Falaschi, C. Olarte, and C. Palamidessi. Abstract interpretation of temporal concurrent constraint programs. *TPLP*, 15(3):312–357, 2015.
12. 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.
13. S. Josep. A vocabulary of program slicing-based techniques. *ACM Comput. Surv.*, 44(3):12:1–12:41, June 2012.
14. Sophia Knight, C. Palamidessi, P. Panangaden, and F. D. Valencia. Spatial and epistemic modalities in constraint-based process calculi. In M. Koutny and I. Ulidowski, editors, *CONCUR*, volume 7454 of *LNCS*, pages 317–332. Springer, 2012.
15. B. Korel and J. Laski. Dynamic program slicing. *Inf. Process. Lett.*, 29(3):155–163, 1988.
16. M. Nielsen, C. Palamidessi, and F. D. Valencia. On the expressive power of temporal concurrent constraint programming languages. In *Proceedings of PPDP'02*, pages 156–167. ACM, 2002.
17. M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nord. J. Comput.*, 9(1):145–188, 2002.
18. 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.
19. C. Olarte, C. Rueda, G. Sarria, M. Toro, and F. D. Valencia. Concurrent constraints models of music interaction. In G. Assayag and C. Truchet, editors, *Constraint Programming in Music*, pages 133–153. Wiley, 2011.
20. C. Olarte, C. Rueda, and F. D. Valencia. Models and emerging trends of concurrent constraint programming. *Constraints*, 18(4):535–578, 2013.
21. C. Olarte and F. D. Valencia. Universal concurrent constraint programing: symbolic semantics and applications to security. In R. L. Wainwright and H. Haddad, editors, *SAC*, pages 145–150. ACM, 2008.
22. Carlos Olarte, Elaine Pimentel, and Vivek Nigam. Subexponential concurrent constraint programming. *Theor. Comput. Sci.*, 606:98–120, 2015.
23. V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
24. V. A. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *J. Symb. Comput.*, 22(5/6):475–520, 1996.
25. 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.
26. E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, 1983.
27. M. Weiser. Program slicing. *IEEE Trans. on Software Engineering*, 10(4):352–357, 1984.