Abstract Diagnosis for Timed Concurrent Constraint programs

MARCO COMINI, LAURA TITOLO
Dipartimento di Matematica e Informatica
University of Udine
Via delle Scienze, 206
33100 Udine, Italy
(*e-mail:* {marco.comini,laura.titolo}@uniud.it)

ALICIA VILLANUEVA*
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València
Camino de Vera s/n
46022 Valencia, Spain
(*e-mail:* villanue@dsic.upv.es)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

The *Timed Concurrent Constraint Language* (*tccp* in short) is a concurrent logic language based on the simple but powerful concurrent constraint paradigm of Saraswat. In this paradigm, the notion of store-as-value is replaced by the notion of store-as-constraint, which introduces some differences w.r.t. other approaches to concurrency.

In this paper, we provide a general framework for the debugging of *tccp* programs. To this end, we first present a new compact, bottom-up semantics for the language that is well suited for debugging and verification purposes in the context of reactive systems. We also provide an abstract semantics that allows us to effectively implement debugging algorithms based on abstract interpretation.

Given a *tccp* program and a behavior specification, our debugging approach automatically detects whether the program satisfies the specification. This differs from other semi-automatic approaches to debugging and avoids the need to provide symptoms in advance. We show the efficacy of our approach by introducing an application example. We choose a specific abstract domain and show how we can detect that a program is erroneous.

KEYWORDS: concurrent constraint paradigm, denotational semantics, abstract diagnosis, abstract interpretation

1 Introduction

Finding program bugs is a long-standing problem in software construction. In the concurrent paradigms, the problem is even worse and the traditional tracing tech-

* This work has been partially supported by the EU (FEDER), the Spanish MICINN under grant TIN2010-21062-C02-02 and by the Universitat Politècnica de València under grant PAID-00-10.

niques are almost useless. There has been a lot of work on algorithmic debugging (Shapiro 1982) for declarative languages, which could be a valid proposal for concurrent paradigms, but little effort has been done for the particular case of the concurrent constraint paradigm (*ccp* in short; (Saraswat 1993)). The *ccp* paradigm is different from other programming paradigms mainly due to the notion of store-as-constraint that substitutes the classical store-as-valuation model. In this way, the languages from this paradigm can easily handle partial information: an underlying constraint system handles constraints on system variables. Within this family, (de Boer et al. 2000) introduced the *Timed Concurrent Constraint Language* (*tccp* in short) by adding to the original *ccp* model the notion of time and the ability to capture the absence of information. With these features, it is possible to specify behaviors typical of reactive systems such as *timeouts* or *preemption* actions, but they also make the language non-monotonic.

In this paper, we develop an abstract diagnosis method for *tccp* using the ideas of the abstract diagnosis framework for logic programming (Comini et al. 1999). This framework, parametric w.r.t. an abstract program property, is based on the use of an abstract immediate consequence operator to identify bugs in logic programs. It can be considered as an extension of algorithmic debugging since there are instances of the framework that deliver the same results. The intuition of the approach is that, given an abstract specification of the expected behavior of the program, one automatically detects the errors in the program. The framework does not require the determination of symptoms in advance. In order to achieve an effective method, abstract interpretation is used to approximate the semantics, thus results may be less precise than those obtained by using the concrete semantics.

The approach of abstract diagnosis for logic programming has been applied to other paradigms (Alpuente et al. 2003; Bacci and Comini 2010; Falaschi et al. 2007). This research revealed that a key point for the efficacy of the resulting debugging methodology is the compactness of the concrete semantics. Thus, in this proposal, much effort has been devoted to the development of a compact concrete semantics for the *tccp* language to start with. The already existing denotational semantics are based on capturing the input-output behavior of the system. However, since we are in a concurrent (reactive) context, we want to analyze and debug infinite computations. Our semantics covers this need and is suitable to be used not only with debugging techniques but also with other verification approaches.

Our new (concrete) compact *compositional* semantics is correct and fully abstract w.r.t. the small-step behavior of *tccp*. It is based on the evaluation of agents over a denotation for a set of process declarations D, obtained as least fixpoint of a (continuous) immediate consequence operator $\mathcal{D}[\![D]\!]$.

Thanks to the compactness of this semantics we can formulate an efficacious debugging methodology based on abstract interpretation which proceeds by approximating the $\mathcal{D}[\![D]\!]$ operator producing an "abstract immediate consequence operator" $\mathcal{D}^{\alpha}[\![D]\!]$. We show that, given the abstract intended specification $\mathcal{S}^{\alpha}$ of the semantics of a program D, we can check the correctness of D by a single application of $\mathcal{D}^{\alpha}[\![D]\!]$ and thus, by a static test, we can determine all the process declarations $d \in D$ which are wrong w.r.t. the considered abstract property.

To our knowledge, in the literature there is only another approach to the debugging problem of *ccp* languages, (Falaschi et al. 2007), which is also based on the abstract diagnosis approach of (Comini et al. 1999). However, they consider a quite different concurrent constraint language without non-monotonic features, which we consider essential to model behaviors of reactive systems.

2 The Timed Concurrent Constraint language

All the languages of the *ccp* paradigm (Saraswat 1993) are parametric w.r.t. a cylindric constraint system. The constraint system handles the data information of the program in terms of constraints. In *tccp*, the computation progresses as the concurrent and asynchronous activity of several agents that can (monotonically) accumulate information in a store, or query some information from that store. Briefly, a cylindric constraint system[1] $\mathbf{C} = \langle \mathcal{C}, \preceq, \otimes, \oplus, tt, \mathit{ff}, \mathit{Var}, \exists \rangle$ is composed of a set of finite constraints $\mathcal{C}$ ordered by $\preceq$, where $\oplus$ and $\otimes$ are the *glb* and *lub*, respectively. tt is the smaller constraint whereas ff is the bigger one. We often use the inverse order $\vdash$ instead of $\preceq$ over constraints. Given a cylindric constraint system $\mathbf{C}$ and a set of process symbols Π, the syntax of agents is given by the following grammar:

$$A ::= \mathsf{skip} \mid \mathsf{tell}(c) \mid \sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i \mid \mathsf{now}(c)\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2 \mid A_1 \parallel A_2 \mid \exists x\, A \mid p(\vec{x})$$

where c and c_i are finite constraints in $\mathcal{C}$, $p \in \Pi$, $x \in \mathit{Var}$ and $\vec{x}$ is the list of variables $x_1, \ldots, x_n$ with $1 \leq i \leq n$, $x_i \in \mathit{Var}$. A *tccp* program P is an object of the form $D.A_0$, where A_0 is an agent, called initial agent, and D is a set of process declarations of the form $p(\vec{x}) :\!- A$ (for some agent A).

The notion of time is introduced by defining a discrete and global clock: it is assumed that the ask and tell agents take one time-unit to be executed. For the operational semantics of the language, the reader can consult (de Boer et al. 2000). Intuitively, the skip agent represents the successful termination of the agent computation. The $\mathsf{tell}(c)$ agent adds the constraint c to the current store and stops at the following time instant. The store is updated by means of the $\otimes$ operator of the constraint system. The choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i$ consults the store and non-deterministically executes (in the following time instant) one of the agents A_i, whose corresponding guard c_i holds in the current store; otherwise, if no guard is satisfied by the store, the agent suspends. The agent $\mathsf{now}(c)\ \mathsf{then}\ A\ \mathsf{else}\ B$ behaves like A (respectively B) if c is (respectively is not) entailed by the store. Note that this agent can process negative information: it can capture when some information is not present in the store since the agent B is executed both when $\neg c$ is satisfied, but also when neither c nor $\neg c$ are satisfied. Moreover, differently from the ask case, it evaluates the guard instantaneously. $A \parallel B$ models the parallel composition of A and B in terms of maximal parallelism (in contrast to the interleaving approach

[1] See (de Boer et al. 2000; Saraswat 1993) for more details on cylindric constraint systems.

of *ccp*), i.e., all the enabled agents of A and B are executed at the same time. The agent $\exists x\, A$ is used to make variable x local to A, by means of the $\exists$ operator of the constraint system. Finally, the agent $p(\vec{x})$ takes from D the declaration of the form $p(\vec{x}) :- A$ and executes A at the following time instant. For the sake of simplicity, we assume that the set D of declarations is closed w.r.t. parameter names.

3 Modeling the small-step operational behavior of *tccp*

In this section, we introduce a denotational semantics that models the small-step behavior of *tccp*. Due to space limitations, in this paper we show the most relevant aspects of both the concrete and the abstract semantics. The complete definitions, as well as the proofs of all the results, can be found in (Comini et al. 2011).

First let us formalize the notion of behavior for a set D of process declarations. Intuitively, it collects all the small-step computations associated to D as the set of (all the prefixes of) the sequences of computational steps, for all possible initial agents and stores.

Definition 1 (*Small-step behavior of declarations*)
Let D be a set of declarations, *Agent* the set of possible agents, and $\rightarrow$ the transition relation given by the operational semantics in (de Boer et al. 2000). The small-step behavior of D is defined as follows:

$$\mathcal{B}^{ss}[\![D]\!] := \bigcup_{\forall c \in \mathcal{C}, \forall A \in Agent} \mathcal{B}[\![D.A]\!]_c$$

where $\mathcal{B}[\![D.A_0]\!]_{c_0} := \{c_0 \cdot \dots \cdot c_n \mid \langle A_0, c_0\rangle \rightarrow \ldots \rightarrow \langle A_n, c_n\rangle\} \cup \{\epsilon\}$. We denote by $\approx_{ss}$ the equivalence relation between declarations induced by $\mathcal{B}^{ss}$, namely $D_1 \approx_{ss} D_2 \Leftrightarrow \mathcal{B}^{ss}[\![D_1]\!] = \mathcal{B}^{ss}[\![D_2]\!]$.

There are many languages where a compact compositional semantics has been founded on collecting the possible traces for the weakest store, since all traces relative to any other initial store can be derived by instance of the formers. However, in *tccp* this does not work since it is not monotonic: if we have all traces for an agent A starting from an initial store c and we execute A with a more instantiated initial store d, then new traces, not instances of the formers, can appear.[2]

Furthermore, note that, since we are interested in a bottom-up approach, we cannot work assuming that we know the initial store. However, when we have to give the semantics of a conditional or choice agent where some guard must be checked, we should take a different execution branch depending on its satisfiability. Our idea is that of associating conditions to computation steps and to collect all possible minimal hypothetical computations.

[2] See (Comini et al. 2011) for some examples.

3.1 The semantic domain

In (de Boer et al. 2000), reactive sequences are used as semantic domain for the top-down semantics.[3] These sequences consists of pairs of stores $\langle c, c' \rangle$ for each time instant meaning that, given the initial store c, the program produces in one time instant the store c'. The store is monotonic, thus c' always contains more (or equal) information than c.

As we have explained before, this information is not enough for a bottom-up approach. The idea is to enrich the reactive sequence so that we keep information about the essential conditions that the store must satisfy in order to make the program proceed. We define a condition η as a pair $\eta = (\eta^+, \eta^-)$ where $\eta^+ \in \mathcal{C}$ (respectively $\eta^- \in \wp(\mathcal{C})$) is called positive (respectively negative) component. A condition is said to be *inconsistent* when its positive component entails any constraint in the negative component or when the positive component is *ff*. Given a store $c \in \mathcal{C}$, we say that c satisfies η (written $c \rhd \eta$) when c entails η^+, $\eta^+ \neq \mathit{ff}$ and c does not entail any constraint from η^-. An inconsistent condition is satisfied by no store, while the pair $(\mathit{tt}, \emptyset)$ is satisfied by any store.

A conditional reactive sequence is a sequence of *conditional tuples*, which can be of two forms: (a) a triple $\eta \rightarrow \langle a,\, b \rangle$ that is used to represent a computational step, i.e., the global store a becomes b at the next time instant only if $a \rhd \eta$, or (b) a construct $\mathit{stutt}(C)$ that models the suspension of the computation due to an ask agent, i.e., it represents the fact that there is no guard in C (the guards of the choice agent) entailed by the current store. We need this construct to distinguish a suspended computation from an infinite loop that does not modify the store.

Our denotations are composed of *conditional reactive sequences*:

Definition 2 (*Conditional reactive sequence*)
A conditional reactive sequence is a sequence of conditional tuples of the form $t_1 \dots t_n \dots$, maybe ended with $\Box$, such that: for each $t_i = \eta_i \rightarrow \langle a_i,\, b_i \rangle$, $b_i \vdash a_i$ for $i \geq 1$, and for each $t_j = \eta_j \rightarrow \langle a_j,\, b_j \rangle$ such that $j > i$, $a_j \vdash b_i$. The empty sequence is denoted with ϵ. $s_1 \cdot s_2$ denotes the concatenation of two conditional reactive sequences s_1, s_2.

A set of conditional reactive sequences is *maximal* if none of its sequences is the prefix of another. By $\mathbb{M}$ we denote the domain of sets of maximal conditional reactive sequences, whose order is induced from its prefix closure, namely $R_1 \sqsubseteq R_2 \Leftrightarrow \mathit{prefix}(R_1) \subseteq \mathit{prefix}(R_2)$. $(\mathbb{M}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ is a complete lattice.

3.2 Agent Semantics Evaluation Function

In order to associate a denotation to a set of process declarations, we need first to define the semantics for agents. Let us first introduce the notion of interpretation.

[3] In a top-down approach, the (initial) current store is propagated, thus decisions regarding the satisfaction or not of a given condition can be taken immediately.

Definition 3 (*Interpretations*)
Let $\mathbb{MGC} := \{p(\vec{x}) \mid p \in \Pi,\ \vec{x}$ are distinct variables$\}$ be the set of most general calls. An *interpretation* is a function $\mathbb{MGC} \to \mathbb{M}$ modulo variance.[4] Two functions $I, J : \mathbb{MGC} \to \mathbb{M}$ are *variants*, denoted by $I \cong J$, if for each $\pi \in \mathbb{MGC}$ there exists a variable renaming ρ such that $(I\pi)\rho = J(\pi\rho)$. The semantic domain $\mathbb{I}$ is the set of all interpretations ordered by the point-wise extension of $\sqsubseteq$.

The application of an interpretation $\mathcal{I}$ to a most general call π, denoted by $\mathcal{I}(\pi)$, is the application $I(\pi)$ of any representative I of $\mathcal{I}$ which is defined exactly on π. For example, if $\mathcal{I} = (\lambda\varphi(x,y).\, \{(tt, \emptyset) \to \langle tt,\, x = y\rangle\})/_{\cong}$ then $\mathcal{I}(\varphi(u,v)) = \{(tt,\emptyset) \to \langle tt,\, u = v\rangle\}$.

The technical core of our semantics definition is the agent evaluation semantic function which, given an agent and an interpretation, builds the maximal conditional reactive sequences of the agent.

Definition 4 (*Agents Semantics*)
Given an agent A and an interpretation $\mathcal{I}$, the semantics $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ is defined by structural induction:

$$\begin{aligned}
&\mathcal{A}[\![\mathsf{skip}]\!]_{\mathcal{I}} = \{\boxtimes\}\\
&\mathcal{A}[\![\mathsf{tell}(c)]\!]_{\mathcal{I}} = \{(tt,\emptyset) \to \langle tt,\, c\rangle \cdot \boxtimes\} && (3.1)\\
&\mathcal{A}[\![\textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\mathcal{I}} = \bigsqcup_{i=1}^{n} \{(c_i,\emptyset) \to \langle c_i,\, c_i\rangle \cdot (s \odot c_i) \mid s \in \mathcal{A}[\![A_i]\!]_{\mathcal{I}}\} \sqcup\\
&\qquad \bigsqcup \{\mathit{stutt}(\cup_{i=1}^{n} c_i) \cdot s \mid s \in \mathcal{A}[\![\textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\mathcal{I}}, \forall i \in [1,n].c_i \neq tt\} && (3.2)\\
&\mathcal{A}[\![\mathsf{now}(d) \text{ then } A \text{ else } B]\!]_{\mathcal{I}} = \{(d,\emptyset) \to \langle d,\, d\rangle \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{I}}\} \sqcup\\
&\qquad \bigsqcup \{(c^{+} \otimes d, c^{-}) \to \langle c \otimes d,\, c' \otimes d\rangle \cdot (s \odot d) \mid (c^{+}, c^{-}) \to \langle c,\, c'\rangle \cdot s \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\\
&\qquad\qquad c \otimes d \rhd (c^{+} \otimes d, c^{-})\} \sqcup\\
&\qquad \bigsqcup \{(d, C) \to \langle d,\, d\rangle \cdot (s \odot d) \mid \mathit{stutt}(C) \cdot s \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\ d \rhd (d, C)\} \sqcup\\
&\qquad \bigsqcup \{(tt, d) \to \langle tt,\, tt\rangle \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} \sqcup\\
&\qquad \bigsqcup \{(c^{+}, c^{-} \cup \{d\}) \to \langle c,\, c'\rangle \cdot s \mid (c^{+}, c^{-}) \to \langle c,\, c'\rangle \cdot s \in \mathcal{A}[\![B]\!]_{\mathcal{I}},\\
&\qquad\qquad c \rhd (c^{+}, c^{-} \cup \{d\})\} \sqcup\\
&\qquad \bigsqcup \{(tt, C \cup \{d\}) \to \langle tt,\, tt\rangle \cdot s \mid \mathit{stutt}(C) \cdot s \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} && (3.3)\\
&\mathcal{A}[\![A \parallel B]\!]_{\mathcal{I}} = \bigsqcup \{s_A \dot{\parallel} s_B \mid s_A \in \mathcal{A}[\![A]\!]_{\mathcal{I}}, s_B \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} && (3.4)\\
&\mathcal{A}[\![\exists_x A]\!]_{\mathcal{I}} = \bigsqcup \{s \in \mathbb{M} \mid \exists s' \in \mathcal{A}[\![A]\!]_{\mathcal{I}} \text{ such that } \exists_x s = \exists_x s', && (3.5)\\
&\qquad\qquad s' \text{ is } x\text{-connected, } s \text{ is } x\text{-invariant}\}\\
&\mathcal{A}[\![p(z)]\!]_{\mathcal{I}} = \bigsqcup \{(tt,\emptyset) \to \langle tt,\, tt\rangle \cdot s \mid s \in \mathcal{I}(p(z))\}
\end{aligned}$$

Let us now illustrate the idea of the semantics. In (3.1), since a tell agent works independently of the current store, we associate condition $(tt, \emptyset)$. The second part

[4] i.e., a family of elements of $\mathbb{M}$ indexed by $\mathbb{MGC}$ modulo variance.

of the conditional tuple states that the constraint c is added during the current computational step. Finally, the computation terminates $\Box$.

The semantics for the non-deterministic choice, described in (3.2) collects for each guard c_i a conditional sequence of the form $(c_i, \emptyset) \rightarrow \langle c_i,\, c_i \rangle \cdot (s \odot c_i)$. The condition states that c_i has to be satisfied by the current store, whereas the pair $\langle c_i, c_i \rangle$ represents the fact that the query to the store does not modify the store. The constraint c_i is in addition propagated (by means of the propagation operator $\odot$ (Comini et al. 2011)) to the sequence s (the continuation of the computation), which belongs to the semantics of A_i. Moreover, we have to model the case when the computation suspends, i.e., when no guard of the agent is satisfied by the current store. Sequences representing this situation are of the form $\mathit{stutt}(\cup_{i=1}^{n}\{c_i\}) \cdot s$ where s is, recursively, an element of the semantics of the choice agent. The only case when we do not include the stuttering sequence is when any of the guards c_i is tt. Note that, due to the partial nature of the constraint system, the fact that the disjunction of the guards is tt is not a sufficient condition to avoid the suspension.

The definition of the conditional agent now is similar to the previous one. However, since it is instantaneous, we have 6 cases depending on the 3 possible heads of the sequences of the semantics of A (respectively B) and on the fact that the guard d is satisfied or not in this instant.

The semantics for the parallel composition of two agents (Equation (3.4)) is defined in terms of an auxiliary commutative operator $\dot{\|}$ (see (Comini et al. 2011)), which intuitively parallely combines the sequences of the two agents.

For the hiding operator (Equation (3.5)), we collect the sequences that satisfy the restrictions regarding the visibility of the hided variables. This is formalized by the notions of x-connected and x-invariant adapted from (de Boer et al. 2000).

Finally, the semantics of the process call $p(\vec{x})$ collects the sequences in the interpretation $\mathcal{I}(p(\vec{x}))$, delayed by one time unit, as stated in the operational semantics.

Let us show an illustrative example. Consider the *tccp* agent $A \equiv \mathsf{ask}(y \geq 0) \rightarrow \mathsf{tell}(z \leq 0)$. The semantics is composed of one sequence representing the case when the guard is satisfied, and the case when the computation suspends.

$$\begin{aligned}\mathcal{A}[\![A]\!]_{\mathcal{I}} = &\{\, (y \geq 0, \emptyset) \rightarrow \langle y \geq 0,\, y \geq 0 \rangle \cdot (tt, \emptyset) \rightarrow \langle y \geq 0,\, y \geq 0 \otimes z \leq 0 \rangle \cdot \Box \} \\ &\cup \{\mathit{stutt}(y \geq 0) \cdot s \mid s \in \mathcal{A}[\![A]\!]_{\mathcal{I}}\}\end{aligned}$$

3.3 Fixpoint Denotations of Declarations

Now we can define the semantics for a set of process declarations D as the fixpoint of the immediate consequences operator $\mathcal{D}[\![D]\!]_{\mathcal{I}} := \lambda p(x). \bigcup_{p(x):-A \in D} \mathcal{A}[\![A]\!]_{\mathcal{I}}$, which is continuous. Thus it has a least fixpoint and we can define the semantics of D as $\mathcal{F}[\![D]\!] = \mathit{lfp}(\mathcal{D}[\![D]\!])$. As an example, in Figure 1 we represent the (infinite) set of traces of $\mathcal{F}[\![\{p(x) :- \exists y\,(\,\mathsf{ask}(y > x) \rightarrow p(x+1) + \mathsf{ask}(y \leq x) \rightarrow \mathsf{skip})\}]\!]$.[5]

[5] For the sake of simplicity, we assume that we can use expressions of the form $x+1$ directly in the arguments of a process call. We can simulate this behavior by writing $\mathsf{tell}(x' = x + 1) \rightarrow p(x')$ (but introducing a delay of one time unit).

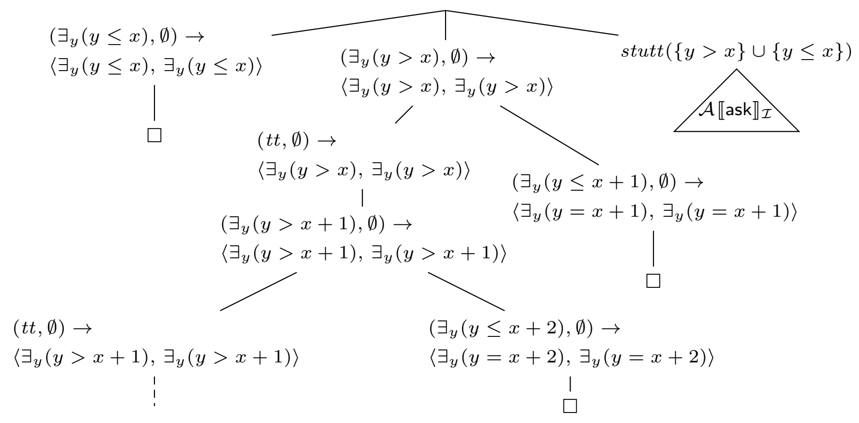
$(\exists_y(y \leq x), \emptyset) \rightarrow \langle \exists_y(y \leq x), \exists_y(y \leq x) \rangle$
$(\exists_y(y > x), \emptyset) \rightarrow \langle \exists_y(y > x), \exists_y(y > x) \rangle$
$stutt(\{y > x\} \cup \{y \leq x\})$
$\mathcal{A}[\![\mathsf{ask}]\!]_{\mathcal{I}}$
$(tt, \emptyset) \rightarrow \langle \exists_y(y > x), \exists_y(y > x) \rangle$
$(\exists_y(y \leq x+1), \emptyset) \rightarrow \langle \exists_y(y = x+1), \exists_y(y = x+1) \rangle$
$(\exists_y(y > x+1), \emptyset) \rightarrow \langle \exists_y(y > x+1), \exists_y(y > x+1) \rangle$
$(tt, \emptyset) \rightarrow \langle \exists_y(y > x+1), \exists_y(y > x+1) \rangle$
$(\exists_y(y \leq x+2), \emptyset) \rightarrow \langle \exists_y(y = x+2), \exists_y(y = x+2) \rangle$

Figure 1. Tree representation of $\mathcal{F}[\![D]\!]$ in the example.

In (Comini et al. 2011) we proved that $D_1 \approx_{ss} D_2$ if and only if $\mathcal{F}[\![D_1]\!] = \mathcal{F}[\![D_2]\!]$ (correctness and full abstraction of $\mathcal{F}$ w.r.t. $\approx_{ss}$).

4 Abstract semantics for *tccp*: the abstraction scheme

In this section, starting from the fixpoint semantics in Section 3, we present an abstract semantics which approximates the observable behavior of the program. Program properties that are of interest are Galois Insertions between the concrete domain and the chosen abstract domain. We assume familiarity with basic results of abstract interpretation (Cousot and Cousot 1979).

We define an abstraction scheme where we define the abstraction of computations, i.e., of maximal sets of conditional reactive sequences, by successive lifting. We start with a function that abstracts the information component of the program, i.e., the store, and then we build the abstraction of conditional tuple, then of conditional reactive sequences and finally of maximal sets.

We start from a upper-approximating function $\tau^+ : \mathcal{C} \rightarrow \hat{\mathcal{C}}$ into an abstract constraint system $\hat{\mathbf{C}} = \langle \hat{\mathcal{C}}, \hat{\preceq}, \hat{\otimes}, \hat{\oplus}, \hat{tt}, \hat{ff}, Var, \hat{\exists} \rangle$, where $\hat{tt}$ and $\hat{ff}$ are the smallest and the greatest abstract constraint, respectively. We often use the inverse relation $\hat{\vdash}$ of $\hat{\preceq}$. We have also a lower-approximating function $\tau^- : \wp(\mathcal{C}) \rightarrow \check{\mathcal{C}}$ into an abstract constraint system $\check{\mathbf{C}} = \langle \check{\mathcal{C}}, \check{\preceq}, \check{\otimes}, \check{\oplus}, \check{tt}, \check{ff}, Var, \check{\exists} \rangle$.

We have two "external" operations $\hat{\times} : \mathcal{C} \times \hat{\mathcal{C}} \rightarrow \hat{\mathcal{C}}$ and $\check{\times} : \mathcal{C} \times \check{\mathcal{C}} \rightarrow \check{\mathcal{C}}$ that update an abstract store with a concrete constraint (coming from the program).

Abstract and concrete constraint systems are related by these conditions:

$$\begin{array}{ll}
c \mathbin{\hat{\times}} \tau^+(a) = \tau^+(c \otimes a) & c \mathbin{\check{\times}} \tau^-(C) = \tau^-(\{c\} \cup C) \\
\tau^+(a \otimes b) = \tau^+(a) \mathbin{\hat{\otimes}} \tau^+(b) & \tau^-(C \cup C') = \tau^-(C) \mathbin{\check{\oplus}} \tau^-(C') \\
a \vdash b \Longrightarrow \tau^+(a) \mathrel{\hat{\vdash}} \tau^+(b) & \tau^-(\{a\}) \mathrel{\check{\vdash}} \tau^-(C) \Longrightarrow \exists c \in C.\, a \vdash c \\
\tau^+(\exists_x a) = \hat{\exists}_x \tau^+(a) & \tau^-(\{\exists_x c \mid c \in C\}) = \check{\exists}_x \tau^-(C)
\end{array}$$

Now, an abstract condition is a pair of the form $(\hat{\eta}, \check{\eta}) \in \hat{\mathcal{C}} \times \check{\mathcal{C}}$. Similarly to the concrete case, given an abstract condition $\tilde{\eta} = (\hat{\eta}, \check{\eta})$ and an abstract store $\hat{a} \in \hat{\mathcal{C}}$, we say that $\hat{a}$ satisfies $\tilde{\eta}$ (written $\hat{a} \mathrel{\tilde{\rhd}} \tilde{\eta}$) when $\hat{\eta} \neq \hat{\mathit{ff}}$ and $\hat{a} \mathrel{\hat{\vdash}} \hat{\eta}$, but $\hat{a} \mathrel{\check{\nvdash}} \check{\eta}$. Given an abstract condition $\tilde{\eta}$, $\hat{a}, \hat{b} \in \hat{\mathcal{C}}$ and $\check{c} \in \check{\mathcal{C}}$, we define an abstract conditional tuple either as a triple $\tilde{\eta} \to \langle \hat{a},\, \hat{b} \rangle^{m}$, such that $\hat{a} \mathrel{\tilde{\rhd}} \tilde{\eta}$, or a construct of the form $\mathit{stutt}^{\alpha}(\check{c})^{m}$, where $m \in \{0, +\infty\}$ indicates how many times the corresponding concrete tuples appear consecutively in the concrete sequence. Given a (concrete) conditional tuple, we define its abstraction α as

$$
\begin{aligned}
&\alpha((\eta^{+}, \eta^{-}) \to \langle a,\, b \rangle) = (\tau^{+}(\eta^{+}), \tau^{-}(\eta^{-})) \to \langle \tau^{+}(a),\, \tau^{+}(b) \rangle^{1} \\
&\alpha(\mathit{stutt}(C)) = \mathit{stutt}^{\alpha}(\tau^{-}(C))^{1}
\end{aligned}
$$

Now, an abstract conditional reactive sequence is a sequence of abstract tuples of the form: $\tilde{t}_1 \ldots \tilde{t}_m \ldots$, maybe ended with $\Box$, such that for each couple of tuples $\tilde{t}_i = \tilde{\eta}_i \to \langle \hat{a}_i,\, \hat{b}_i \rangle^{m_i}$, $\tilde{t}_j = \tilde{\eta}_j \to \langle \hat{a}_j,\, \hat{b}_j \rangle^{m_j}$, and $i \neq j$, $\tilde{t}_i \neq \tilde{t}_j$. The natural number associated to each abstract conditional tuple is needed to keep synchronization among processes due to the particularly strong synchronization properties of the language, as already noticed in (Alpuente et al. 2005).

The abstraction $\alpha(s)$ of a sequence of conditional tuples s is defined by structural induction on the form of tuples t. It collapses all the computation steps (conditional tuples) that, after abstraction, coincide. The base cases are $\alpha(\epsilon) = \epsilon$ and $\alpha(\Box) = \Box$.

$$
\alpha(t \cdot r) := \begin{cases}
\tilde{\eta} \to \langle \hat{a},\, \hat{b} \rangle^{m+1} \cdot \tilde{r} & \text{if } \alpha(t) = \tilde{\eta} \to \langle \hat{a},\, \hat{b} \rangle^{1},\ \alpha(r) = \tilde{\eta} \to \langle \hat{a},\, \hat{b} \rangle^{m} \cdot \tilde{r} \\
\mathit{stutt}^{\alpha}(\check{c})^{m+1} \cdot \tilde{r} & \text{if } \alpha(t) = \mathit{stutt}^{\alpha}(\check{c})^{1},\ \alpha(r) = \mathit{stutt}^{\alpha}(\check{c})^{m} \cdot \tilde{r} \\
\alpha(t) \cdot \alpha(r) & \text{otherwise}
\end{cases}
$$

We extend the definition to sets of conditional sequences in the natural way. We denote by $\mathbb{A}$ the domain $\alpha(\mathbb{M})$ of the sets of abstract conditional reactive sequences. By adjunction we derive the concretization function γ such that $(\mathbb{M}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathbb{A}, \leq, \bigvee, \bigwedge, \bot, \top)$, where $a \leq a' \iff \gamma(a) \sqsubseteq \gamma(a')$.

This abstraction can be systematically lifted to the domain of interpretations: $\mathbb{I} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} [PA \to \mathbb{A}]$ so that we can derive the optimal abstraction of $\mathcal{D}[\![D]\!]$ simply as $\mathcal{D}^{\alpha}[\![D]\!] := \alpha \circ \mathcal{D}[\![D]\!] \circ \gamma$. The abstract interpretation theory ensures that $\mathcal{F}^{\alpha}[\![D]\!] := \mathcal{D}^{\alpha}[\![D]\!] \uparrow \omega$ is the best correct approximation of $\mathcal{F}[\![D]\!]$.

It turns out that $\mathcal{D}^{\alpha}[\![D]\!]_{\mathcal{I}^{\alpha}} = \lambda p(x). \bigcup_{p(x) :- A \in D} \mathcal{A}^{\alpha}[\![A]\!]_{\mathcal{I}^{\alpha}}$, where $\mathcal{A}^{\alpha}[\![\cdot]\!]_{\mathcal{I}^{\alpha}}$ is defined by structural induction on the syntax in a similar way as the concrete version. Given the similarity to the concrete case, in the following we describe only some cases to illustrate the use of the over- and lower-approximations and the two "external" functions, for full details consult (Comini et al. 2011).

The $\tilde{\odot}$ operator is the abstract counterpart of the concrete version. The semantics for the tell agent just applies the abstraction to the only concrete sequence, thus: $\mathcal{A}^{\alpha}[\![\mathsf{tell}(c)]\!]_{\mathcal{I}^{\alpha}} = \{(\hat{\mathit{tt}}, \check{\mathit{ff}}) \to \langle \hat{\mathit{tt}},\, \tau^{+}(c) \rangle^{1} \cdot \Box\}$.

For the now semantics, we only show the general case when the condition holds, and the general case when it does not hold:

$$
\begin{aligned}
&\mathcal{A}^{\alpha}[\![\mathsf{now}(d)\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_{\mathcal{I}^{\alpha}} = \\
&\quad \{(d \mathbin{\hat{\times}} \hat{\eta}, \check{\eta}) \rightarrow \langle d \mathbin{\hat{\times}} \hat{a},\, d \mathbin{\hat{\times}} \hat{b}\rangle^{n} \cdot (d \mathbin{\tilde{\odot}} \tilde{s}) \mid \\
&\qquad (\hat{\eta}, \check{\eta}) \rightarrow \langle \hat{a},\, \hat{b}\rangle^{n} \cdot \tilde{s} \in \mathcal{A}^{\alpha}[\![A]\!]_{\mathcal{I}^{\alpha}} \text{ and } d \mathbin{\hat{\times}} \hat{a} \mathrel{\tilde{\rhd}} (d \mathbin{\hat{\times}} \hat{\eta}, \check{\eta})\} \\
&\quad \cup \ldots \\
&\quad \cup \{(\hat{\eta}, d \mathbin{\check{\times}} \check{\eta}) \rightarrow \langle \hat{a},\, \hat{b}\rangle^{1} \cdot (\hat{\eta}, \check{\eta}) \rightarrow \langle \hat{a},\, \hat{b}\rangle^{n} \cdot \tilde{s} \mid \\
&\qquad (\hat{\eta}, \check{\eta}) \rightarrow \langle \hat{a},\, \hat{b}\rangle^{n+1} \cdot \tilde{s} \in \mathcal{A}^{\alpha}[\![B]\!]_{\mathcal{I}^{\alpha}} \text{ and } \hat{a} \mathrel{\tilde{\rhd}} (\hat{\eta}, d \mathbin{\check{\times}} \check{\eta})\} \\
&\quad \cup \ldots
\end{aligned}
$$

5 Abstract diagnosis of timed concurrent constraint programs

Now, following the ideas of (Comini et al. 1999), we define the abstract diagnosis of *tccp*. The framework of abstract diagnosis (Comini et al. 1999) comes from the idea of considering the abstract versions of Park's Induction Principle[6]. It can be considered as an extension of declarative debugging since there are instances of the framework that deliver the same results. However, in the general case, diagnosing w.r.t. *abstract* properties relieves the user from having to specify in excessive detail the program behavior (which could be more error-prone than the coding itself).

Let us now introduce the workset of abstract diagnosis. Having chosen a property of the computation α of interest (an instance of the abstraction scheme of Section 4), given a set of declarations D and $\mathcal{S}^{\alpha} \in \mathbb{A}$, which is the specification of the intended behavior of D w.r.t. the property α, we say that

1. D is (abstractly) *partially correct* w.r.t. $\mathcal{S}^{\alpha}$ if $\alpha(\mathcal{F}[\![D]\!]) \leq \mathcal{S}^{\alpha}$.
2. D is (abstractly) *complete* w.r.t. $\mathcal{S}^{\alpha}$ if $\mathcal{S}^{\alpha} \leq \alpha(\mathcal{F}[\![D]\!])$.
3. D is *totally correct* w.r.t. $\mathcal{S}^{\alpha}$, if it is partially correct and complete.

It is worth noting that in this setting the user can only reason in terms of the properties of the expected concrete semantics without being concerned with (approximate) abstract computations.

The *diagnosis* determines the "originating" symptoms and, in the case of incorrectness, the relevant process declaration in the program. This is captured by the definitions of *abstractly incorrect process declaration* and *abstract uncovered element*.

Definition 5
Let D be a set of declarations, R a process declaration and $\{e\}, \mathcal{S}^{\alpha} \in \mathbb{A}$.
R is *abstractly incorrect* w.r.t. $\mathcal{S}^{\alpha}$ if $\mathcal{D}^{\alpha}[\![\{R\}]\!]_{\mathcal{S}^{\alpha}} \not\leq \mathcal{S}^{\alpha}$.
e is an *uncovered element* w.r.t. $\mathcal{S}^{\alpha}$ if $\{e\} \leq \mathcal{S}^{\alpha}$ and $\{e\} \wedge \mathcal{D}^{\alpha}[\![D]\!]_{\mathcal{S}^{\alpha}} = \bot$.

Informally, R is abstractly incorrect if it derives a wrong abstract element from the

[6] A concept of formal verification that is undecidable in general.

intended semantics. e is uncovered if all process declarations cannot derive it from the intended semantics.

It is worth noting that correctness and completeness are defined in terms of $\alpha(\mathcal{F}[\![D]\!])$, i.e., in terms of abstraction of the concrete semantics. Thus, the abstract version of algorithmic debugging (Shapiro 1982), which is based on symptoms (i.e., deviations between $\alpha(\mathcal{F}[\![D]\!])$ and $\mathcal{S}^{\alpha}$), would require the construction of $\alpha(\mathcal{F}[\![D]\!])$ and therefore a fixpoint computation. On the other hand, abstractly incorrect process declarations and abstract uncovered elements are defined directly in terms of *just one* application of $\mathcal{D}^{\alpha}[\![D]\!]$ to $\mathcal{S}^{\alpha}$. The issue of the precision of the abstract semantics becomes therefore relevant in establishing the relation between the two concepts. i.e., of proving which is the relation between abstractly incorrect process declarations and abstract uncovered elements on one side, and correctness and completeness, on the other side.

Theorem 1

1. If there are no abstractly incorrect process declarations in D, then D is partially correct w.r.t. $\mathcal{S}^{\alpha}$.
2. Let D be partially correct w.r.t. $\mathcal{S}^{\alpha}$. If D has abstract uncovered elements then D is not complete.

The results when applying the diagnosis w.r.t. approximate properties may be weaker than those that can be achieved on concrete domains just because of approximation. Abstract incorrect process declarations are in general just a warning about a possible source of errors. Because of the approximation, it can happen that a (concretely) correct declaration is abstractly incorrect. However, as shown by the following theorem, all concrete errors are detected, as they lead to an abstract incorrectness or abstract uncovered.[7]

Theorem 2
Let r be a process declaration, $\mathcal{S}$ a concrete specification.

1. If $\mathcal{D}[\![\{r\}]\!]_{\mathcal{S}} \not\sqsubseteq \mathcal{S}$ and $\alpha(\mathcal{D}[\![\{r\}]\!]_{\mathcal{S}}) \not\leq \alpha(\mathcal{S})$ then r is abstractly incorrect w.r.t. $\alpha(\mathcal{S})$.
2. If there exists an abstract uncovered element a w.r.t. $\alpha(\mathcal{S})$, such that $\gamma(a) \sqsubseteq \mathcal{S}$ and $\gamma(\bot) = \bot$, then there exists a concrete uncovered element e w.r.t. $\mathcal{S}$ (i.e., $e \sqsubseteq \mathcal{S}$ and $e \sqcap \mathcal{D}[\![D]\!]_{\mathcal{S}} = \bot$).

There is property of our proposal which is particularly useful for applications: it can be used with partial specifications and also with partial programs. Obviously, one cannot detect errors in process declarations involving functions which have not been specified. But for the process declarations that involve only functions with a specification, the check can be made, even if the whole program has not been written yet. This includes the possibility of applying our "local" method to all parts of a program not involving constructs which we cannot handle (yet). With other "global" approaches such programs could not be checked at all.

[7] The reviewer can find proofs of these results in appendix.

The abstract domain for abstract sequences that we have defined is not noetherian due to the use of the index in each tuple. However, we cannot get rid of it since it is needed to keep synchronization among parallel processes. Nevertheless, if we use an abstract domain for the constraint system which is noetherian, since the store evolves monotonically, it holds that the number of conditional tuples that can appear in a sequence is finite. This property states the kind of specifications that we will be able to handle in an efficacious way.

5.1 Examples of application of the framework

Let us now show some illustrative examples of the approach. The first example shows the new ability of our approach that can deal with the constructors that introduce the non-monotonic behavior of the system, in particular the now agent.

Example 1

We model a (simplified) *time-out*(n) process that checks for, at most, n times units if the system emits a signal telling that the process evolves normally $(system = ok)$. When the signal arrives, the system emits the fact that there is no alert $(alert = no)$[8]. If the time limit is reached, the system should set the signal *alert* to *yes*. Let d_0, d_n, d_{action} be the following declarations:

$$\begin{aligned}
\textit{time-out}(0)\!:\!-\; & \mathsf{now}(system = ok)\ \mathsf{then}\ action\ \mathsf{else}\\
& \mathsf{ask}(tt) \to \textit{time-out}(0)\\
\textit{time-out}(n)\!:\!-\; & \mathsf{now}(system = ok)\ \mathsf{then}\ action\ \mathsf{else}\\
& \mathsf{ask}(tt) \to \textit{time-out}(n-1)\\
action\!:\!-\; & \mathsf{tell}(alert = no)
\end{aligned}$$

In this case, due to the simplicity of the constraint system, the abstract versions coincide with the concrete one and the two external functions are the $\hat{\oplus}$ and $\check{\oplus}$ operators. Let us now consider the following specification. For d_0 we expect that, if the *ok* signal is present, then it ends with an $alert = no$ signal, otherwise the alert is emitted. The specification for d_n is similar, but we add n sequences, since we have the possibility that the signal arrives at each time instant before n.

$$\begin{aligned}
\mathcal{S}^{\alpha}(\textit{time-out}(0)) = \; & \{\, (system = ok, \check{f\!f}) \to \langle system = ok,\ system = ok\rangle^{1} \cdot\\
& \quad (\hat{tt}, \check{f\!f}) \to \langle system = ok,\ system = ok \mathbin{\hat{\otimes}} alert = no\rangle^{1} \cdot \Box \}\\
& \cup \{(\hat{tt}, \{system = ok\}) \to \langle \hat{tt},\ \hat{tt}\rangle^{1} \cdot (\hat{tt}, \check{f\!f}) \to \langle \hat{tt},\ alert = yes\rangle^{1} \cdot \Box\}\\
\mathcal{S}^{\alpha}(\textit{time-out}(n)) = \; & \{\, (\hat{tt}, \{system = ok\}) \to \langle \hat{tt},\ \hat{tt}\rangle^{m} \cdot\\
& \quad (system = ok, \check{f\!f}) \to \langle system = ok,\ system = ok\rangle^{1} \cdot\\
& \quad (\hat{tt}, \check{f\!f}) \to \langle system = ok,\ system = ok \mathbin{\hat{\otimes}} alert = no\rangle^{1} \cdot \Box \mid 0 \le m < n\}\\
& \cup\ \{(\hat{tt}, \{system = ok\}) \to \langle \hat{tt},\ \hat{tt}\rangle^{n+1} \cdot (\hat{tt}, \check{f\!f}) \to \langle \hat{tt},\ alert = yes\rangle^{1} \cdot \Box\}\\
\mathcal{S}^{\alpha}(action) = \; & \{(\hat{tt}, \check{f\!f}) \to \langle \hat{tt},\ alert = no\rangle^{1} \cdot \Box\}
\end{aligned}$$

$\mathcal{D}^{\alpha}[\![\{d_0\}]\!]_{\mathcal{S}^{\alpha}}$ is:

[8] The classical timeout would restart the countdown by recursively calling *time-out*(n).

$$
\begin{aligned}
&\{(\mathit{system} = \mathit{ok}, \check{\mathit{ff}}) \to \langle \mathit{system} = \mathit{ok},\ \mathit{system} = \mathit{ok}\rangle^1 \cdot \\
&\quad (\hat{\mathit{tt}}, \check{\mathit{ff}}) \to \langle \mathit{system} = \mathit{ok},\ \mathit{system} = \mathit{ok} \mathbin{\hat{\otimes}} \mathit{alert} = \mathit{no}\rangle^1 \cdot \Box\} \\
&\cup\ \{(\hat{\mathit{tt}}, \{\mathit{system} = \mathit{ok}\}) \to \langle \hat{\mathit{tt}},\ \hat{\mathit{tt}}\rangle^1 \cdot (\mathit{system} = \mathit{ok}, \check{\mathit{ff}}) \to \langle \mathit{system} = \mathit{ok},\ \mathit{system} = \mathit{ok}\rangle^1 \cdot \\
&\qquad (\hat{\mathit{tt}}, \check{\mathit{ff}}) \to \langle \mathit{system} = \mathit{ok},\ \mathit{system} = \mathit{ok} \mathbin{\hat{\otimes}} \mathit{alert} = \mathit{no}\rangle^1 \cdot \Box\} \\
&\cup\ \{(\hat{\mathit{tt}}, \{\mathit{system} = \mathit{ok}\}) \to \langle \hat{\mathit{tt}},\ \hat{\mathit{tt}}\rangle^2 \cdot (\hat{\mathit{tt}}, \check{\mathit{ff}}) \to \langle \hat{\mathit{tt}},\ \mathit{alert} = \mathit{no}\rangle^1 \cdot \Box\}
\end{aligned}
$$

Due to the last sequence, $\mathcal{D}^{\alpha}[\![\{d_0\}]\!]_{\mathcal{S}^{\alpha}} \not\leq \mathcal{S}^{\alpha}$, so we conclude that d_0 is (abstractly) incorrect. In fact, the recursive call in the else branch of the declaration is not what we expect as behavior of a time-out. The correct declaration d_0' replaces the recursive call by $\mathsf{tell}(\mathit{alert} = \mathit{yes})$. $\mathcal{D}^{\alpha}[\![\{d_0'\}]\!]_{\mathcal{S}^{\alpha}} \leq \mathcal{S}^{\alpha}$, thus d_0' is abstractly correct.

Our second example shows a system already studied in (Falaschi et al. 2007). We have adapted it to the *tccp* language so that we show how our semantics differs from theirs. It is important to remark that this example does not handle negative information, differently from Example 1. In this case, we check whether an error signal arrives, in other words, *someone* must tell us that such situation occurs. In the timeout example, we are able to detect that something goes wrong due to the absence of the *ok* signal, not thanks to a specific signal.

Example 2
Let D be a set containing the following declarations (d_1 and d_2, respectively). The idea of the system is to control, at each time instant, if a failure signal has been arrived. In that case, an action is taken (for instance just a constraint *stop* is added).

$$
\begin{aligned}
\mathit{control}(i,o) \mathrel{:-} \exists o', i'.\ & \mathsf{now}(i = [\mathit{fail}|\_])\ \mathsf{then}\ (\mathsf{tell}(i = [\mathit{fail}|i'])\ \|\ \mathit{action}(o,o')) \\
& \qquad\qquad\qquad \mathsf{else\ skip} \\
& \|\ \mathsf{ask}(\mathit{tt}) \to \mathit{control}(i',o') \\
\mathit{action}(o,o') \mathrel{:-} & \ \mathsf{tell}(o = [\mathit{stop}|o'])
\end{aligned}
$$

The concrete domain for the constraint system is composed by the elements *fail*, *stop*, *tt* and *ff*. The abstract setting is similar to the one in the previous example. Due to the monotonicity of the store, we have to use streams to model the *imperative-style* variables (de Boer et al. 2000). Following this idea, the abstraction for concrete streams is defined as the last instantiated value in the stream. The concretization of one stream is defined as all the concrete streams whose last value is a concretization of the abstract one. We write a dot on a predicate symbol (e.g. $\dot{=}$) to denote that we want to check it for the last instantiated value of a stream.

Let us now check that the *action* process finishes in one time instant. To this end, we define the following specification: $\mathcal{S}^{\alpha}(\mathit{action}(x_1,x_2)) = \{(\hat{\mathit{tt}}, \check{\mathit{ff}}) \to \langle \hat{\mathit{tt}},\ x_1 \dot{=} \mathit{stop}\rangle^1 \cdot \Box\}$. If we compute one iteration of the semantic operator on the specification, we get $\mathcal{D}^{\alpha}[\![\{d_2\}]\!]_{\mathcal{S}^{\alpha}} = \{(\hat{\mathit{tt}}, \check{\mathit{ff}}) \to \langle \hat{\mathit{tt}},\ \tau^{+}(o = [\mathit{stop}|o'])\rangle^1 \cdot \Box\} = \{(\hat{\mathit{tt}}, \check{\mathit{ff}}) \to \langle \hat{\mathit{tt}},\ o \dot{=} \mathit{stop}\rangle^1 \cdot \Box\}$, thus we conclude that the declaration d_2 is correct w.r.t $\mathcal{S}^{\alpha}$.

Let us now consider the *control* process and its specification in order to show the relevance of the property captured by the abstraction:

$$\begin{aligned}\mathcal{S}^{\alpha}(\mathit{control}(f,s)) = \{ & (\hat{tt}, \{f \doteq \mathit{fail}\}) \rightarrow \langle \hat{tt},\ \hat{tt} \rangle^{n} \cdot \\ & (f \doteq \mathit{fail}, \check{\mathit{ff}}) \rightarrow \langle f \doteq \mathit{fail},\ f \doteq \mathit{fail} \mathbin{\hat{\otimes}} \hat{\exists}\, f'(f' \doteq \mathit{true}) \rangle^{1} \cdot \\ & (\hat{tt}, \check{\mathit{ff}}) \rightarrow \langle f \doteq \mathit{fail} \mathbin{\hat{\otimes}} \hat{\exists}\, f'(f' \doteq \mathit{true}),\ f \doteq \mathit{fail} \mathbin{\hat{\otimes}} \hat{\exists}\, f'(f' \doteq \mathit{true}) \mathbin{\hat{\otimes}} s \doteq \mathit{stop} \rangle^{1} \cdot \Box \mid n \geq 0 \} \\ \cup \{ & (\hat{tt}, \{f \doteq \mathit{fail}\}) \rightarrow \langle \hat{tt},\ \hat{tt} \rangle^{+\infty} \}\end{aligned}$$

Note that this specification is infinite. In order to make the abstract diagnosis process effective, one solution would be to use our framework with a (much concrete) depth-k abstraction (similar to what is done in (Falaschi et al. 2007)) in order to check only up to a given time instant k. This it is not equivalent to model-check (for instance) an equivalent temporal property (written in some temporal logic). We remark that this problem is natural when we try to specify properties corresponding to an abstract property that is not Noetherian.

Finally, we show how one can work with the abstraction of the constraint system, and also how we can take advantage of our abstract domain.

Example 3
Let us consider a system with a single declaration and the abstraction of the constraint system that abstracts integer variables to a (simplified) interval-based domain with abstract values $\{\top, \mathrm{pos}_x, \mathrm{neg}_x, x{>}10, x{\leq}10, \bot\}$.

$$\begin{aligned} p(x) :- \ & \mathsf{now}(x \dot{>} 0)\ \mathsf{then}\ \exists x'\ \mathsf{tell}(x = [\_|x']) \parallel \mathsf{tell}(x' = [x+1|\_]) \parallel p(x') \\ & \mathsf{else}\ \exists x''\ \mathsf{tell}(x = [\_|x'']) \parallel \mathsf{tell}(x'' = [x-1|\_]) \parallel p(x'') \end{aligned}$$

We define the following intended specification to specify that, (a) if the parameter is greater than 10, then the last value of the stream (written $\dot{x}_1$ will always be greater than 10; (b) if the parameter is negative, then the value is always negative

$$\begin{aligned}\mathcal{S}^{\alpha}(p(x_1) = & \{(\dot{x}_1{>}10, \check{\mathit{ff}}) \rightarrow \langle \dot{x}_1{>}10,\ \dot{x}_1{>}10 \rangle^{+\infty}\} \\ & \cup \{(\mathrm{neg}_{\dot{x}_1}), \check{\mathit{ff}}) \rightarrow \langle \mathrm{neg}_{\dot{x}_1}),\ \mathrm{neg}_{\dot{x}_1} \rangle^{+\infty}\}\end{aligned}$$

This is a partial specification of the system behavior, but as we have already explained, this is not a problem in the abstract diagnosis approach. The two abstract sequences are representing infinite computations thanks to the $+\infty$ index in the last tuple. Therefore, although we cannot tackle infinite specifications, finite specifications that represent infinite computations can be considered and effectively handled. In fact, we can compute $\mathcal{D}^{\alpha}[\![\{d\}]\!]_{\mathcal{S}^{\alpha}}$:

$$\begin{aligned} & \{ \{(\dot{x}{>}10, \check{\mathit{ff}}) \rightarrow \langle \dot{x}{>}10,\ \dot{x}{>}10 \rangle^{1} \cdot (\dot{x}{>}10, \check{\mathit{ff}}) \rightarrow \langle \dot{x}{>}10,\ \dot{x}{>}10 \rangle^{+\infty}\} \\ & \quad \cup \{(\mathrm{neg}_{\dot{x}}), \check{\mathit{ff}}) \rightarrow \langle \mathrm{neg}_{\dot{x}},\ \mathrm{neg}_{\dot{x}} \rangle^{1} \cdot (\mathrm{neg}_{\dot{x}}, \check{\mathit{ff}}) \rightarrow \langle \mathrm{neg}_{\dot{x}},\ \mathrm{neg}_{\dot{x}} \rangle^{+\infty}\}\} \\ & = \\ & \{ \{(\dot{x}{>}10, \check{\mathit{ff}}) \rightarrow \langle \dot{x}{>}10,\ \dot{x}{>}10 \rangle^{+\infty}\} \cup \{(\mathrm{neg}_{\dot{x}}, \check{\mathit{ff}}) \rightarrow \langle \mathrm{neg}_{\dot{x}},\ \mathrm{neg}_{\dot{x}} \rangle^{+\infty}\}\}\end{aligned}$$

We can see that in the first computed sequence, although the initial condition for the now agent is $x \dot{>} 0$, when using the interpretation of $p(x)$ and combining it with the (partial) computed sequence, we merge the conditions and we get the more restrictive constraint of $\dot{x}{>}10$. We can see that $\mathcal{D}^{\alpha}[\![\{d\}]\!]_{\mathcal{S}^{\alpha}} \leq \mathcal{S}^{\alpha}$, thus we conclude that the declaration is correct w.r.t. $\mathcal{S}^{\alpha}$.

6 Related Work

A top-down (big-step) denotational semantics for *tccp* is defined in (de Boer et al. 2000) for terminating computations. In that work, a terminating computation is both, a computation that reaches a point in which no agents are pending to be executed, and also a computation that suspends since there is no enough information in the store to make the choice agents evolve. Our semantics is a bottom-up (small-step) denotational semantics that models infinite computations, and also distinguishes the two kinds of terminating computations aforementioned. Conceptually, a suspended computation has not completely finished its execution, and, in some cases, it could be a symptom of a system error. Thus, the new semantics is well suited to handle, not only functional systems (where an input-output semantics makes sense), but also reactive systems.

In (Falaschi et al. 2007), a first approach to the declarative debugging of a *ccp* language is presented. However, it does not cover the particular extra difficulty of the non-monotonicity, common to all timed concurrent constraint languages. As we have said, this ability is crucial in order to model specific behaviors of reactive systems, such as timeouts or preemption actions. This is the main reason why our abstract (and concrete) semantics are significantly different from (Falaschi et al. 2007) and from formalizations for other declarative languages.

The idea of using two different mechanisms for dealing with positive and negative information in our abstraction scheme is inspired by (Alpuente et al. 2005), where two entailment relations were used to get a correct approach, based on source-to-source transformation, to the abstraction of *tccp* programs.

7 Conclusion and Future Work

We have presented a new compact, bottom-up semantics for the *tccp* language which is correct and fully abstract w.r.t. the behavior of the language. This semantics is well suited for debugging and verification purposes in the context of reactive systems.

Then, an abstract semantics that is able to specify (a kind of) infinite computations is presented. It is based on the abstraction of computation sequences by using two functions that satisfy some properties in order to guarantee correctness. All our examples satisfy those conditions. The abstract semantics keeps the synchronization among parallel computations, which is a particular difficulty of the *tccp* language. As already noticed in (Alpuente et al. 2005), the loss of synchronization in other *ccp* languages just implies a loss of precision, but in the case of *tccp*, due to the maximal parallelism, it would imply a loss of correctness.

Finally, we have adapted the abstract diagnosis approach to the *tccp* language employing the new semantics as basis. We have presented some illustrative examples that show the new features of our approach w.r.t. other paradigms, and also its limitations.

As future work, we intend to work on abstractions of our semantics to domains of temporal logic formulas, in order to be able to specify safety and/or liveness properties, and to compare its models w.r.t. the program semantics.

References

Alpuente, M., Comini, M., Escobar, S., Falaschi, M., and Lucas, S. 2003. Abstract Diagnosis of Functional Programs. In *Logic Based Program Synthesis and Transformation – 12th International Workshop, LOPSTR 2002, Revised Selected Papers*, M. Leuschel, Ed. Lecture Notes in Computer Science, vol. 2664. Springer-Verlag, Berlin, 1–16.

Alpuente, M., Gallardo, M., Pimentel, E., and Villanueva, A. 2005. A Semantic Framework for the Abstract Model Checking of tccp Programs. *Theoretical Computer Science 346,* 1, 58–95.

Bacci, G. and Comini, M. 2010. Abstract Diagnosis of First Order Functional Logic Programs. In *Logic-Based Program Synthesis and Transformation – 20th International Symposium – Pre-Proceedings*, M. Alpuente, Ed. RISC-Linz Report Series, vol. 10-14. 58–72.

Comini, M., Levi, G., Meo, M. C., and Vitiello, G. 1999. Abstract Diagnosis. *Journal of Logic Programming 39,* 1-3, 43–93.

Comini, M., Titolo, L., and Villanueva, A. 2011. A Compact Goal-Independent Bottom-Up Fixpoint Modeling the Behavior of tccp. Tech. Rep. DIMI-UD/01/2011/RR, Dipartimento di Matematica e Informatica, U. di Udine. Available at URL: http://www.dimi.uniud.it/comini/Papers/.

Cousot, P. and Cousot, R. 1979. Systematic Design of Program Analysis Frameworks. In *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, San Antonio, Texas, January 29–31.* ACM Press, New York, NY, USA, 269–282.

de Boer, F. S., Gabbrielli, M., and Meo, M. C. 2000. A Timed Concurrent Constraint Language. *Information and Computation 161,* 1, 45–83.

Falaschi, M., Olarte, C., Palamidessi, C., and Valencia, F. 2007. Declarative diagnosis of temporal concurrent constraint programs. In *Proceedings of the 23rd International Conference on Logic Programming (ICLP'07).* Lecture Notes in Computer Science, vol. 4670. Springer-Verlag, 271–285.

Saraswat, V. A. 1993. *Concurrent Constraint Programming.* The MIT Press, Cambridge, Mass.

Shapiro, E. Y. 1982. Algorithmic Program Debugging. In *Proceedings of Ninth Annual ACM Symp. on Principles of Programming Languages.* ACM Press, New York, NY, USA, 412–531.

Appendix A Proofs (For referees)

Proof of Theorem 1

Point 1 By hypothesis $\forall r \in D.\ \mathcal{D}^\alpha[\![\{r\}]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$. Hence $\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$, i.e., $\mathcal{S}^\alpha$ is a pre-fixpoint of $\mathcal{D}^\alpha[\![D]\!]$. Since $\alpha(\mathcal{F}[\![D]\!]) \leq \mathcal{F}^\alpha[\![D]\!] = \mathit{lfp}\, \mathcal{D}^\alpha[\![D]\!]$, by Knaster–Tarski's Theorem $\alpha(\mathcal{F}[\![D]\!]) \leq \mathcal{F}^\alpha[\![D]\!] \leq \mathcal{S}^\alpha$. The thesis follows by definition of correctness.

Point 2 By construction $\alpha \circ \mathcal{D}[\![D]\!] \circ \gamma \leq \mathcal{D}^\alpha[\![D]\!]$, hence $\alpha \circ \mathcal{D}[\![D]\!] \circ \gamma \circ \alpha \leq \mathcal{D}^\alpha[\![D]\!] \circ \alpha$. Since $\mathit{id} \sqsubseteq \gamma \circ \alpha$, it holds that $\alpha \circ \mathcal{D}[\![D]\!] \leq \alpha \circ \mathcal{D}[\![D]\!] \circ \gamma \circ \alpha$ and $\alpha \circ \mathcal{D}[\![D]\!] \leq \mathcal{D}^\alpha[\![D]\!] \circ \alpha$. Hence,

$$\begin{array}{ll}
\alpha(\mathcal{F}[\![D]\!]) = & [\,\text{since } \mathcal{F}[\![D]\!] \text{ is a fixpoint}\,] \\
\alpha(\mathcal{D}[\![D]\!]_{\mathcal{F}[\![D]\!]}) \leq & [\,\text{by } \alpha \circ \mathcal{D}[\![D]\!] \leq \mathcal{D}^\alpha[\![D]\!] \circ \alpha\,] \\
\mathcal{D}^\alpha[\![D]\!]_{\alpha(\mathcal{F}[\![D]\!])} \leq & [\,\text{since } \mathcal{D}^\alpha[\![D]\!] \text{ is monotone and } D \text{ is partial correct}\,] \\
\mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha} &
\end{array}$$

Now, if D has an abstract uncovered element e i.e., $\{e\} \leq \mathcal{S}^\alpha$ and $\{e\} \wedge \mathcal{D}^\alpha[\![D]\!]_{\mathcal{S}^\alpha} = \bot$, then $\{e\} \wedge \alpha(\mathcal{F}[\![D]\!]) = \bot$ and $\mathcal{S}^\alpha \not\leq \alpha(\mathcal{F}[\![D]\!])$. The thesis follows from definition of completeness.

□

Proof of Theorem 2

Point 1 Since $\mathcal{S} \sqsubseteq \gamma \circ \alpha(\mathcal{S})$, by monotonicity of α and the correctness of $\mathcal{D}^\alpha[\![\{r\}]\!]$, it holds that $\alpha(\mathcal{D}[\![\{r\}]\!]_\mathcal{S}) \leq \alpha(\mathcal{D}[\![\{r\}]\!]_{\gamma \circ \alpha(\mathcal{S})}) \leq \mathcal{D}^\alpha[\![\{r\}]\!]_{\alpha(\mathcal{S})}$. By hypothesis $\alpha(\mathcal{D}[\![\{r\}]\!]_\mathcal{S}) \not\leq \alpha(\mathcal{S})$, therefore $\mathcal{D}^\alpha[\![\{r\}]\!]_{\alpha(\mathcal{S})} \not\leq \alpha(\mathcal{S})$, since $\alpha(\mathcal{D}[\![\{r\}]\!]_\mathcal{S}) \leq \mathcal{D}^\alpha[\![\{r\}]\!]_{\alpha(\mathcal{S})}$. The thesis holds by Definition 5.

Point 2 By hypothesis $a \leq \alpha(\mathcal{S})$ and $a \wedge \mathcal{D}^\alpha[\![D]\!]_{\alpha(\mathcal{S})} = \bot$. Hence $\gamma(a) \sqcap \gamma(\mathcal{D}^\alpha[\![D]\!]_{\alpha(\mathcal{S})}) = \bot$ since $\gamma(\bot) = \bot$ and γ preserves greatest lower bounds. By construction $\mathcal{D}^\alpha[\![D]\!] = \alpha \circ \mathcal{D}[\![D]\!] \circ \gamma$, thus $\gamma(a) \sqcap \gamma(\alpha(\mathcal{D}[\![D]\!]_{\gamma(\alpha(\mathcal{S}))})) = \bot$. Since $\mathit{id} \sqsubseteq \gamma \circ \alpha$ and by monotonicity of $\mathcal{D}[\![D]\!]$, $\gamma(a) \sqcap \mathcal{D}[\![D]\!]_\mathcal{S} = \bot$. By hypothesis $\gamma(a) \sqsubseteq \mathcal{S}$ hence $\gamma(a)$ is a concrete uncovered element.

□