Titolo, L.; Villanueva García, A.; Comini, M. (2011). Abstract diagnosis for timed concurrent constraint programs. Theory and Practice of Logic Programming. 11:487-502. doi:10.1017/S1471068411000135.

# Abstract Diagnosis for Timed Concurrent Constraint programs*†

M. Comini, L. Titolo and A. Villanueva

**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 two illustrative examples. 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 techniques are almost useless. There has been a lot of work on algorithmic debugging [1] for declarative languages, which could be a valid proposal for concurrent paradigms, but little effort has been done for the particular case of

*This work has been published in Theory and Practice of Logic Programming, volume 11, pages 487502, 2011. Cambridge University Press.

†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.

the concurrent constraint paradigm (*ccp* in short; [2]). The *ccp* paradigm is different from other programming paradigms mainly due to the notion of store-as-constraint that replaces the classical store-as-valuation model. In this way, the languages from this paradigm can easily deal with partial information: an underlying constraint system handles constraints on system variables. Within this family, [3] 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 [4]. 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 [5, 6, 7]. 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, monotone) 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 the declarations $D$, 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, [7], which is also based on the abstract

diagnosis approach of [4]. 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

The *tccp* language is particularly suitable to specify both reactive and time critical systems. As the other languages of the *ccp* paradigm [2], it is 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] $\mathbf{C} = \langle \mathcal{C}, \preceq, \otimes, \oplus, tt, ff, Var, \exists \rangle$ 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 smallest constraint whereas $ff$ is the largest one. We often use the inverse order $\vdash$ (called *entailment*) instead of $\preceq$ over constraints. *Var* is a denumerable set of variables and $\exists$ existentially quantifies variables over constraints (the so called cylindric operator).

Given a cylindric constraint system $\mathbf{C}$ and a set of process symbols $\Pi$, the syntax of agents is given by the following grammar:

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

where $c$ and $c_i$ are finite constraints in $\mathcal{C}$, $p \in \Pi$, $x \in Var$ and $\vec{x}$ is a list of variables $x_1, \ldots, x_n$ with $1 \leq i \leq n$, $x_i \in Var$. 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 [3]. 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. It takes one time-unit, thus the constraint $c$ is visible to other agents from the following time instant. The store is updated by means of the $\otimes$ operator of the constraint system. The choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ consults the store and nondeterministically executes (at 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 in the current time instant like $A$ (respectively $B$) if $c$ is (respectively is not) satisfied by the store. The satisfaction is checked by using the $\vdash$ operator of the constraint system. Note that this agent can process negative information: it can capture when some information is not present in the store since the

[1] See [3, 2] for more details on cylindric constraint systems.

agent $B$ is executed both when $\neg c$ is satisfied, but also when neither $c$ nor $\neg c$ are satisfied. $A \parallel B$ models the parallel composition of $A$ and $B$ in terms of maximal parallelism (in contrast to the interleaving approach 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 $A$. To this end, it uses the $\exists$ operator of the constraint system. Finally, the agent $p(\vec{x})$ takes from $D$ a 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 concrete semantics and the most relevant aspects of the abstract one. The missing definitions, as well as the proofs of all the results, can be found in [8].

Let us formalize the notion of behavior for a set $D$ of process declarations. 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 [3]. 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 $\mathcal{B}[\![D.A]\!]_c := \{c \cdot c_1 \cdot \dots \cdot c_n \mid \langle A, c\rangle \rightarrow \langle A_1, c_1\rangle \rightarrow \dots \rightarrow \langle A_n, c_n\rangle\} \cup \{\epsilon\}$. We denote by $\approx_{ss}$ the equivalence relation between declarations induced by $\mathcal{B}^{ss}$, namely $D_1 \approx_{ss} D_2 \Leftrightarrow \mathcal{B}^{ss}[\![D_1]\!] = \mathcal{B}^{ss}[\![D_2]\!]$.*

The pair $\langle A_i, c_i\rangle$ denotes a configuration where $A_i$ is the agent to be executed, and $c_i$ the store at that computation step. Thus, the small-step behavior is the set of sequences of stores that are computed by the operational semantics of the language.

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. In *tccp*, this does not work since the language 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.

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 define the semantics of a conditional or choice agent where some guard must be checked, we should consider different execution branches depending on the guard satisfiability. To deal with all these particular features, our idea is that of associating conditions to computation steps, and to collect all possible minimal hypothetical computations.

## 3.1 The semantic domain

In [3], reactive sequences are used as semantic domain for the top-down semantics. These sequences are composed of a pair of stores $\langle c, c' \rangle$ for each time instant meaning that, given the 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.[2] Our idea is to enrich the reactive sequence notion 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* $\eta$ 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 is $ff$ or when it entails any constraint in the negative component. Given a store $c \in \mathcal{C}$, we say that $c$ satisfies $\eta$ (written $c \rhd \eta$) when $c$ entails $\eta^+$, $\eta^+ \neq ff$ and $c$ does not entail any constraint from $\eta^-$. An inconsistent condition is satisfied by no store, while the pair $(tt, \emptyset)$ is satisfied by any store.

A *conditional reactive sequence* is a sequence of *conditional tuples*, which can be of two forms: (i) 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 (ii) a construct $stutt(C)$ 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 $\epsilon$. $s_1 \cdot s_2$ 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 $R_1 \sqsubseteq R_2 \Leftrightarrow prefix(R_1) \subseteq prefix(R_2)$. $(\mathbb{M}, \sqsubseteq, \bigsqcup, \bigsqcap, \bot, \top)$ is a complete lattice.

[2] 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.

## 3.2 Semantics Evaluation Function for Agents

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

**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*[3]. *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* $\rho$ *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 $\pi$, denoted by $\mathcal{I}(\pi)$, is the application $I(\pi)$ of any representative $I$ of $\mathcal{I}$ which is defined exactly on $\pi$. 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 semantics evaluation 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:*

$$\mathcal{A}[\![\mathsf{skip}]\!]_{\mathcal{I}} = \{\Box\}$$

$$\mathcal{A}[\![\mathsf{tell}(c)]\!]_{\mathcal{I}} = \{(tt,\emptyset) \to \langle tt,\ c\rangle \cdot \Box\} \tag{1}$$

$$\mathcal{A}[\![\textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\mathcal{I}} = \bigsqcup_{i=1}^{n}\{(c_i,\emptyset) \to \langle c_i,\ c_i\rangle \cdot (c_i \odot s) \mid s \in \mathcal{A}[\![A_i]\!]_{\mathcal{I}}\} \sqcup$$
$$\bigsqcup\{stutt(\cup_{i=1}^{n} c_i) \cdot s \mid s \in \mathcal{A}[\![\textstyle\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\mathcal{I}}, \forall i \in [1,n].c_i \neq tt\} \tag{2}$$

$$\mathcal{A}[\![\mathsf{now}(d)\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_{\mathcal{I}} = \{(d,\emptyset) \to \langle d,\ d\rangle \cdot \Box \mid \Box \in \mathcal{A}[\![A]\!]_{\mathcal{I}}\} \sqcup$$
$$\bigsqcup\{(c^+ \otimes d, c^-) \to \langle c \otimes d,\ c' \otimes d\rangle \cdot (d \odot s) \mid (c^+,c^-) \to \langle c,\ c'\rangle \cdot s \in \mathcal{A}[\![A]\!]_{\mathcal{I}},$$
$$c \otimes d \rhd (c^+ \otimes d, c^-)\} \sqcup$$
$$\bigsqcup\{(d,C) \to \langle d,\ d\rangle \cdot (d \odot s) \mid stutt(C) \cdot s \in \mathcal{A}[\![A]\!]_{\mathcal{I}},\ d \rhd (d,C)\} \sqcup$$
$$\bigsqcup\{(tt,d) \to \langle tt,\ tt\rangle \cdot \Box \mid \Box \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} \sqcup$$
$$\bigsqcup\{(c^+, c^- \cup \{d\}) \to \langle c,\ c'\rangle \cdot s \mid (c^+,c^-) \to \langle c,\ c'\rangle \cdot s \in \mathcal{A}[\![B]\!]_{\mathcal{I}},$$
$$c \rhd (c^+, c^- \cup \{d\})\} \sqcup$$
$$\bigsqcup\{(tt, C \cup \{d\}) \to \langle tt,\ tt\rangle \cdot s \mid stutt(C) \cdot s \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} \tag{3}$$

$$\mathcal{A}[\![A \parallel B]\!]_{\mathcal{I}} = \bigsqcup\{s_A \dot{\|} s_B \mid s_A \in \mathcal{A}[\![A]\!]_{\mathcal{I}}, s_B \in \mathcal{A}[\![B]\!]_{\mathcal{I}}\} \tag{4}$$

$$\mathcal{A}[\![\exists x\, A]\!]_{\mathcal{I}} = \bigsqcup\{s \in \mathbb{M} \mid \exists s' \in \mathcal{A}[\![A]\!]_{\mathcal{I}} \text{ such that } \exists_x s = \exists_x s',\ s' \text{ is } x\text{-connected, } s \text{ is } x\text{-invariant}\} \tag{5}$$

$$\mathcal{A}[\![p(z)]\!]_{\mathcal{I}} = \bigsqcup\{(tt,\emptyset) \to \langle tt,\ tt\rangle \cdot s \mid s \in \mathcal{I}(p(z))\}$$

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

Let us now illustrate the idea of the semantics. The tell agent works independently of the current store, thus in (1), the conditional reactive sequence starts with a conditional tuple composed by the condition $(tt, \emptyset)$, which is always satisfied, and a second part that says that the constraint $c$ is added during the first computational step; afterwards, the computation terminates with $\Box$.

The semantics for the non-deterministic choice (2), collects for each guard $c_i$ a conditional sequence of the form $(c_i, \emptyset) \rightarrow \langle c_i,\ c_i\rangle \cdot (c_i \odot s)$. 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 propagated into the sequence $s$ (the continuation of the computation which belongs to the semantics of $A_i$) by means of the propagation operator that (consistently) adds a given constraint to the stores appearing in a sequence:

$$h \odot s = \begin{cases} \eta \rightarrow \langle a \otimes h,\, b \otimes h\rangle \cdot (h \odot s') & \text{if } s = \eta \rightarrow \langle a,\, b\rangle \cdot s',\ \eta^{+} \otimes h \neq \mathit{ff}, \\ & b \otimes h \neq \mathit{ff}, a \otimes h \rhd \eta \\ \eta \rightarrow \langle a \otimes h,\, \mathit{ff}\rangle & \text{if } s = \eta \rightarrow \langle a,\, b\rangle \cdot s',\ \eta^{+} \otimes h \neq \mathit{ff}, \\ & b \otimes h = \mathit{ff}, a \otimes h \rhd \eta \\ \mathit{stutt}(\eta^{-}) \cdot (h \odot s') & \text{if } s = \mathit{stutt}(\eta^{-}) \cdot s' \\ s & \text{if } s = \epsilon \text{ or } s = \Box \end{cases}$$

In addition, 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 one 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 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 the current time instant.

The semantics for the parallel composition of two agents (4), is defined in terms of an auxiliary commutative operator $\dot{\|}$ which combines the sequences of the two agents:

$$
s_A \dot{\|} s_B = \begin{cases}
(\eta \otimes_c \delta) \to \langle a \otimes c,\, b \otimes d \rangle \cdot (d \odot s'_A) \dot{\|} (b \odot s'_B) & \text{if } s_A = \eta \to \langle a,\, b \rangle \cdot s'_A, \\
& \quad s_B = \delta \to \langle c,\, d \rangle \cdot s'_B, \\
& \quad a \otimes c \rhd (\eta \otimes_c \delta), b \otimes d \neq \mathit{ff} \\
(\eta \otimes_c \delta) \to \langle a \otimes c,\, \mathit{ff} \rangle & \text{if } s_A = \eta \to \langle a,\, b \rangle \cdot s'_A, \\
& \quad s_B = \delta \to \langle c,\, d \rangle \cdot s'_B, \\
& \quad a \otimes c \rhd (\eta \otimes_c \delta), b \otimes d = \mathit{ff} \\
(\eta^+, \eta^- \cup \delta^-) \to \langle a,\, b \rangle \cdot s'_A \dot{\|} (b \odot s'_B) & \text{if } \; s_A = \eta \to \langle a,\, b \rangle \cdot s'_A, \\
& \quad s_B = \mathit{stutt}(\delta^-) \cdot s'_B, \\
& \quad a \rhd (\eta^+, \eta^- \cup \delta^-) \\
\mathit{stutt}(\eta^- \cup \delta^-) \cdot s'_A \dot{\|} s'_B & \text{if } s'_A = \mathit{stutt}(\eta^-) \cdot s'_A, \\
& \quad s'_B = \mathit{stutt}(\delta^-) \cdot s'_B \\
s_A & \text{if } s_B = \epsilon \text{ or } s_B = \Box
\end{cases}
$$

For the hiding operator (5), we collect the sequences that satisfy the restrictions regarding the visibility of the hided variables. In particular, a conditional reactive sequence $s = t_1 \dots t_n \dots$ is *x-connected* when (1) if $t_1 = \eta_1 \to \langle a_1,\, b_1 \rangle$ then $\exists_x a_1 = a_1$ and (2) for each $t_i = \eta_i \to \langle a_i,\, b_i \rangle$ and $t_{i+1} = \eta_{i+1} \to \langle a_{i+1},\, b_{i+1} \rangle$, with $i > 1$, $\exists_x a_{i+1} \otimes b_i = a_{i+1}$. A conditional reactive sequence $s = t_1 \dots t_n \dots$ is *x-invariant* if for each computational step $t_i = \eta_i \to \langle a_i, b_i \rangle$, it holds that $b_i = \exists_x b_i \otimes a_i$.

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 $A \equiv \mathsf{ask}(y \geq 0) \to \mathsf{tell}(z \leq 0)$. The semantics is composed of two sequences:

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

## 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 $\mathcal{D}[\![D]\!]_{\mathcal{I}} := \lambda p(x). \bigsqcup_{p(x) :- A \in D} \mathcal{A}[\![A]\!]_{\mathcal{I}}$, 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 $\mathcal{F}[\![\{p(x) :- \exists y\, (\, \mathsf{ask}(y > x) \to p(x+1) + \mathsf{ask}(y \leq x) \to \mathsf{skip})\}]\!]$.[4]

In [8] we have proven 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] 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 $\mathsf{tell}(x' = x+1) \to p(x')$ (but introducing a delay of one time unit).

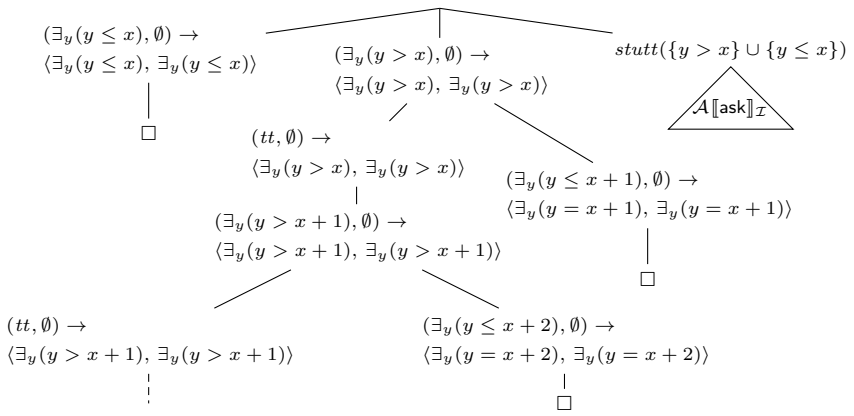


Figure 1: Tree representation of $\mathcal{F}[\![D]\!]$ in the example.

# 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 [9].

We define an abstraction scheme where we develop 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 semantics, i.e., the store; then we build the abstraction of conditional tuples; then of conditional reactive sequences and, finally, of maximal sets.

We start from an upper-approximating function $\tau^+ : \mathcal{C} \to \hat{\mathcal{C}}$ 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 $\tau^- : \wp(\mathcal{C}) \to \check{\mathcal{C}}$ into an abstract constraint system $\check{\mathbf{C}} = \langle \check{\mathcal{C}}, \check{\preceq}, \check{\otimes}, \check{\oplus}, \check{tt}, \check{ff}, Var, \check{\exists} \rangle$. This second function is needed to (correctly) deal with the negative part of conditions.

We have two "external" operations $\hat{\times} : \mathcal{C} \times \hat{\mathcal{C}} \to \hat{\mathcal{C}}$ and $\check{\times} : \mathcal{C} \times \check{\mathcal{C}} \to \check{\mathcal{C}}$ that update an abstract store with a concrete constraint (coming from the program). In addition, a "bridge" relation $\tilde{\vdash} \in \hat{\mathcal{C}} \times \check{\mathcal{C}}$ decides if an upper-abstract constraint is consistent with a lower-abstract constraint. Abstract and concrete constraint systems are related by these conditions:

$$
\begin{array}{ll}
c \mathbin{\hat{\times}} \tau^+(a) = \tau^+(c \otimes a) & c \mathbin{\check{\times}} \tau^-(C) = \tau^-(\{c\} \cup C) \\
\tau^+(a \otimes b) = \tau^+(a) \mathbin{\hat{\otimes}} \tau^+(b) & \tau^-(C \cup C') = \tau^-(C) \mathbin{\check{\oplus}} \tau^-(C') \\
a \vdash b \Longrightarrow \tau^+(a) \mathbin{\hat{\vdash}} \tau^+(b) & \tau^-(\{a\}) \mathbin{\check{\vdash}} \tau^-(C) \Longrightarrow \exists c \in C.\, a \vdash c \\
\tau^+(\exists_x a) = \hat{\exists}_x \tau^+(a) & \tau^-(\{\exists_x c \mid c \in C\}) = \check{\exists}_x \tau^-(C) \\
\forall c \in C.\, a \not\vdash c \Longrightarrow \tau^+(a) \mathbin{\tilde{\not\vdash}} \tau^-(C) &
\end{array}
$$

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 $\hat{a} \mathbin{\tilde{\rhd}} \tilde{\eta}$) when $\hat{\eta} \neq \hat{ff}$ and $\hat{a} \mathbin{\hat{\vdash}} \hat{\eta}$, but $\hat{a} \mathbin{\tilde{\not\vdash}} \check{\eta}$. Given an abstract condition $\tilde{\eta}$, $\hat{a}, \hat{b} \in \hat{\mathcal{C}}$ and $\check{a} \in \check{\mathcal{C}}$, an abstract conditional tuple is either a triple $\tilde{\eta} \to \langle \hat{a},\, \hat{b} \rangle^m$, such that $\hat{a} \mathbin{\tilde{\rhd}} \tilde{\eta}$, or a construct of the form $\mathit{stutt}(\check{a})^m$, where $m \in \{0, +\infty\}$ states how many times the corresponding tuples appear consecutively in the sequence. Given a (concrete) conditional tuple $t$, we define its abstraction $\alpha(t)$ as

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

Now, an abstract conditional reactive sequence is a sequence of *different* abstract tuples $\tilde{t}_1 \ldots \tilde{t}_m \ldots$, maybe ended with $\Box$. 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 [10].

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

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

We extend this 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 $\gamma$ 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: $\mathbb{I} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} [\mathbb{MGC} \to \mathbb{A}]$ 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 $\mathcal{D}^\alpha[\![D]\!]_{\mathcal{I}^\alpha} = \lambda p(x).\bigvee_{p(x):-A\in D} \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha}$, 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 two cases in order to illustrate the use of the upper- and lower-approximations (for full details consult [8]). The semantics for the tell agent just applies the abstraction to the only concrete sequence, thus: $\mathcal{A}^\alpha[\![\mathsf{tell}(c)]\!]_{\mathcal{I}^\alpha} = \{(\hat{tt}, \hat{ff}) \to \langle \hat{tt},\, \tau^+(c)\rangle^1 \cdot \Box\}$. For the now semantics, we only show the general case when the condition holds, and the general case when it does not hold:

$$\begin{array}{l}\mathcal{A}^\alpha[\![\mathsf{now}(d)\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_{\mathcal{I}^\alpha} = \\ \quad \{(d \hat{\times} \hat{\eta}, \check{\eta}) \to \langle d \hat{\times} \hat{a},\, d \hat{\times} \hat{b}\rangle^n \cdot (d \tilde{\odot} \tilde{s}) \mid (\hat{\eta}, \check{\eta}) \to \langle \hat{a},\, \hat{b}\rangle^n \cdot \tilde{s} \in \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha}, d \hat{\times} \hat{a} \tilde{\rhd} (d \hat{\times} \hat{\eta}, \check{\eta})\} \\ \quad \vee \ldots \vee \\ \quad \{(\hat{\eta}, d \check{\times} \check{\eta}) \to \langle \hat{a},\, \hat{b}\rangle^1 \cdot (\hat{\eta}, \check{\eta}) \to \langle \hat{a},\, \hat{b}\rangle^n \cdot \tilde{s} \mid (\hat{\eta}, \check{\eta}) \to \langle \hat{a},\, \hat{b}\rangle^{n+1} \cdot \tilde{s} \in \mathcal{A}^\alpha[\![B]\!]_{\mathcal{I}^\alpha}, \hat{a} \tilde{\rhd} (\hat{\eta}, d \check{\times} \check{\eta})\} \\ \quad \vee \ldots\end{array}$$

the $\tilde{\odot}$ operator is the abstract counterpart of the concrete version.

# 5 Abstract diagnosis of timed concurrent constraint programs

Now, following the ideas of [4], we define the abstract diagnosis of *tccp*. The framework of abstract diagnosis [4] comes from the idea of considering the abstract versions of Park's Induction Principle[5]. It can be considered as an extension of declarative debugging since there are instances of the framework that deliver the same results. 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 $\alpha$ 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 $\alpha$, 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.

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*:

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

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

It is worth noting that the notions of correctness and completeness are defined in terms of $\alpha(\mathcal{F}[\![D]\!])$, i.e., in terms of abstraction of the concrete semantics. The abstract version of algorithmic debugging [1], which is based on symptoms (i.e., deviations between $\alpha(\mathcal{F}[\![D]\!])$ and $\mathcal{S}^\alpha$), requires the construction of $\alpha(\mathcal{F}[\![D]\!])$ and therefore a fixpoint computation. In contrast, the notions of abstractly incorrect process declarations and abstract uncovered elements are defined in terms of *just one* application of $\mathcal{D}^\alpha[\![D]\!]$ to $\mathcal{S}^\alpha$. The issue of the precision of the abstract semantics is specially relevant in establishing the relation between the two concepts (i.e., the relation between abstractly incorrect process declarations and abstract uncovered elements on one side, and abstract partial correctness and completeness, on the other side).[6]

**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.*

When applying the diagnosis w.r.t. approximate properties, the results 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.

**Theorem 2** *Let $r$ be a process declaration and $\mathcal{S}$ 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 $e \sqcap \mathcal{D}[\![D]\!]_{\mathcal{S}} = \bot$).*

It is particularly useful for applications the fact that our proposal can be used with partial specifications and also with partial programs. Obviously, one cannot detect errors in process declarations involving processes which have not been specified, but for the process declarations that involve processes that have

[6] Proofs are available at `http://www.dimi.uniud.it/comini/Papers`.

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.

It is worthy to note that, even for a noetherian abstract constraint system $\hat{\mathbf{C}}$, the domain of abstract sequences defined above is not—in general—noetherian, due to the use of the index in each tuple (we cannot get rid of it since it is needed to keep synchronization among parallel processes). This means that our current proposal cannot be used for static program analysis, unless we resort to use widening operators. However (for noetherian abstract constraint systems) our abstract diagnosis is effective since specifications have to be abstractions of some concrete semantics and, since the store evolves monotonically, it holds that the number of conditional tuples that can appear in an abstract sequence is, thus, finite.

## 5.1 Examples of application of the framework

Let us now show two illustrative examples of the approach. The first example shows the new ability of our approach: that of dealing 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)*[7]*. Let* $d_0$, $d_n$, $d_{action}$ *be the following declarations:*

$$\text{time-out}(0) :- \mathsf{now}(system = ok) \text{ then } action \text{ else } (\mathsf{ask}(tt) \rightarrow \text{time-out}(0))$$
$$\text{time-out}(n) :- \mathsf{now}(system = ok) \text{ then } action \text{ else } (\mathsf{ask}(tt) \rightarrow \text{time-out}(n-1))$$
$$action :- \mathsf{tell}(alert = no)$$

*When the time limit is reached (declaration* $d_0$*), the system should set the signal alert to* yes *(*$\mathsf{tell}(alert = no)$*). However, we have introduced an error in the program, calling the process recursively instead:* time-out$(0)$.

*Due to the simplicity of the constraint system, the abstract domain 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 an alert should be emitted. This is represented by two possible sequences, one with a condition where system = ok, and a second one when system = ok is absent (this is a sequence that reasons with the* absence *of information).*

$$\begin{aligned}\mathcal{S}^{\alpha}(\text{time-out}(0)) = \{ &(system = ok, \check{ff}) \rightarrow \langle system = ok,\ system = ok \rangle^1 \cdot \\ &(\hat{tt}, \check{ff}) \rightarrow \langle system = ok,\ system = ok \hat{\otimes} alert = no \rangle^1 \cdot \Box\} \\ \cup \{&(\hat{tt}, \{system = ok\}) \rightarrow \langle \hat{tt},\ \hat{tt} \rangle^1 \cdot (\hat{tt}, \check{ff}) \rightarrow \langle \hat{tt},\ alert = yes \rangle^1 \cdot \Box\}\end{aligned}$$

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

*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 $n$.*

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

$$\mathcal{S}^\alpha(action) = \{(\hat{tt}, \check{ff}) \to \langle \hat{tt},\ alert = no\rangle^1 \cdot \Box\}$$

*Now, when we compute $\mathcal{D}^\alpha[\![\{d_0\}]\!]_{\mathcal{S}^\alpha}$ we have:*

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

*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. This is due to the recursive call in the else branch of the declaration. If we fix the program replacing $d_0$ by $d_0'$ where the recursive call is replaced by* $\mathsf{tell}(alert = yes)$*, then $\mathcal{D}^\alpha[\![\{d_0'\}]\!]_{\mathcal{S}^\alpha} \leq \mathcal{S}^\alpha$, thus $d_0'$ is abstractly correct.*

In [7] it was studied an example where a *control* process checks whether a *failure* signal arrives to the system. The most important point that differs from the *timeout* example is that, in the *control* case, someone has to explicitly tell the system that an error has occurred. Instead, in the *timeout* example, the system is able to act (and maybe recover) when it detects that something that should have happened, hadn't. In other words, the *control* example does not handle *absence* of information, since non-monotonic operators are not considered there. We have implemented the example in *tccp* and we have checked that the same results can be achieved in our framework if we apply the same abstraction they use (a $depth(k)$ abstraction).

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

**Example 2** *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 $\{\top, pos_x, neg_x, x{>}10, x{\leq}10, \bot\}$.*

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

*Due to the monotonicity of the store, we have to use streams (written in a list-fashion way) to model the* imperative-style *variables [3]. In this way, variable $x$*

*in the program above is a stream that is updated with different values during the execution. Following this idea, the abstraction for concrete streams is defined as the (abstracted) 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.*

*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 $\dot{x}$) will always be greater than 10; (b) if the parameter is negative, then the value is always negative*

$$\mathcal{S}^\alpha(p(x_1) = \{(x_1 \dot{>} 10, \check{ff}) \to \langle x \dot{>} 10,\, x \dot{>} 10\rangle^{+\infty}\} \cup \{(\mathrm{neg}_{\dot{x}}, \check{ff}) \to \langle \mathrm{neg}_{\dot{x}},\, \mathrm{neg}_{\dot{x}}\rangle^{+\infty}\}$$

*The two abstract sequences represent infinite computations thanks to the $+\infty$ index in the last tuple. In other words, finite specifications that represent infinite computations can be considered and effectively handled. In fact, we can compute $\mathcal{D}^\alpha[\![\{d\}]\!]_{\mathcal{S}^\alpha}$:*

$$\begin{aligned}
&\{\,\{(\mathrm{neg}_{\dot{x}}, \check{ff}) \to \langle \mathrm{pos}_{\dot{x}},\, \mathrm{pos}_{\dot{x}}\rangle^1 \cdot (\mathrm{pos}_{\dot{x}} \,\tilde{\odot}\, (\dot{x}{>}10, \check{ff}) \to \langle \dot{x}{>}10,\, \dot{x}{>}10\rangle^{+\infty})\} \\
&\quad \cup \{(\mathrm{neg}_{\dot{x}}, \check{ff}) \to \langle \mathrm{neg}_{\dot{x}},\, \mathrm{neg}_{\dot{x}}\rangle^1 \cdot (\mathrm{neg}_{\dot{x}}, \check{ff}) \to \langle \mathrm{neg}_{\dot{x}},\, \mathrm{neg}_{\dot{x}}\rangle^{+\infty}\}\} \\
&= \\
&\{\,\{(\mathrm{neg}_{\dot{x}}, \check{ff}) \to \langle \mathrm{pos}_{\dot{x}},\, \mathrm{pos}_{\dot{x}}\rangle^1 \cdot \overbrace{(\mathrm{pos}_{\dot{x}} \,\hat{\otimes}\, \dot{x}{>}10}^{\mathrm{pos}_{\dot{x}}}, \check{ff}) \to \langle \mathrm{pos}_{\dot{x}} \,\hat{\otimes}\, \dot{x}{>}10,\, \mathrm{pos}_{\dot{x}} \,\hat{\otimes}\, \dot{x}{>}10\rangle^{+\infty})\} \\
&\quad \cup \{(\mathrm{neg}_{\dot{x}}, \check{ff}) \to \langle \mathrm{neg}_{\dot{x}},\, \mathrm{neg}_{\dot{x}}\rangle^1 \cdot (\mathrm{neg}_{\dot{x}}, \check{ff}) \to \langle \mathrm{neg}_{\dot{x}},\, \mathrm{neg}_{\dot{x}}\rangle^{+\infty}\}\} \\
&= \\
&\{\,\{(\mathrm{pos}_{\dot{x}}, \check{ff}) \to \langle \mathrm{pos}_{\dot{x}},\, \mathrm{pos}_{\dot{x}}\rangle^{+\infty}\} \cup \{(\mathrm{neg}_{\dot{x}}, \check{ff}) \to \langle \mathrm{neg}_{\dot{x}},\, \mathrm{neg}_{\dot{x}}\rangle^{+\infty}\}\}
\end{aligned}$$

*The third equality holds because* $\mathrm{pos}_{\dot{x}}$ *entails* $x \dot{>} 10$*, so the merge of the two constraints will be equal to* $\mathrm{pos}_{\dot{x}}$*.*

*Since $\mathcal{D}^\alpha[\![\{d\}]\!]_{\mathcal{S}^\alpha} \not\leq \mathcal{S}^\alpha$ we can conclude that $d$ is an incorrect declaration w.r.t. $\mathcal{S}^\alpha$. In addition, we can notice that $\mathcal{S}^\alpha$ contain an uncovered element that is a sequence that cannot be derived by the semantics operator.*

# 6 Related Work

A top-down (big-step) denotational semantics for *tccp* is defined in [3] 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 [7], 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 [7] 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 [10]. There, a framework for the abstract model checking of *tccp* programs based on a source-to-source transformation is defined. In particular, it is defined a transformation from a *tccp* program $P$ into a *tccp* program $\bar{P}$ that represents a correct abstraction of the original one (in the sense that the semantics of $P$ are included in the semantics of $\bar{P}$). Instead, we define an abstract semantics for the language. The upper- and lower-approximated versions of the entailment relation are used in order to keep $\bar{P}$ correct, but also precise enough.

# 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. The idea of using conditions in order to have a correct bottom-up semantics can be also applied to other non-monotonic languages such as, for example, ntcc in the *ccp* paradigm [11] or Linda in the imperative (coordination) paradigm [12].

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 [10], 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 two illustrative examples to show the new features of our approach w.r.t. other paradigms.

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. Another interesting aspect is to study if a general framework for the proposed methodology can be defined in order to apply it to other languages.

# References


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

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

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

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

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

[6] G. Bacci and M. Comini. Abstract Diagnosis of First Order Functional Logic Programs. In M. Alpuente, editor, *Logic-based Program Synthesis and Transformation, 20th International Symposium*, volume 6564 of *Lecture Notes in Computer Science*, pages 208–226, Berlin, 2011. Springer-Verlag.

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

[8] M. Comini, L. Titolo, and A. Villanueva. A Compact Goal-Independent Bottom-Up Fixpoint Modeling the Behavior of tccp. Technical Report DIMI-UD/01/2011/RR, Dipartimento di Matematica e Informatica, U. di Udine, 2011. `http://www.dimi.uniud.it/comini/Papers/`.

[9] P. Cousot and R. Cousot. 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*, pages 269–282, New York, NY, USA, 1979. ACM Press.

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

[11] Catuscia Palamidessi and Frank D. Valencia. A Temporal Concurrent Constraint Programming Calculus. In *7th International Conference on Principles and Practice of Constraint Programming (CP'01)*, volume 2239 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 2001.

[12] D. Gelernter. Generative Communication in Linda. *ACM Trans. Program. Lang. Syst. (TOPLAS)*, 7(1):80–113, 1985.