Abstract analysis of universal properties for *tccp*

Marco Comini
DIMI, Università degli Studi di Udine,
`marco.comini@uniud.it`

María del Mar Gallardo, Laura Titolo
LCC, Universidad de Málaga[*]
{`gallardo,laura.titolo`}`@lcc.uma.es`

Alicia Villanueva
DSIC, Universitat Politècnica de València[†]
`villanue@dsic.upv.es`

2016-08-13

Abstract

The Timed Concurrent Constraint Language (*tccp*) is a time extension of the concurrent constraint paradigm of Saraswat. *tccp* was defined to model reactive systems, where infinite behaviors arise naturally. In previous works, a semantic framework and abstract diagnosis method for the language has been defined.

On the basis of that semantic framework, this paper proposes an abstract semantics that, together with a widening operator, is suitable for the definition of different analyses for *tccp* programs. The abstract semantics is correct and can be represented as a finite graph where each node represents a hypothetical computational step of the program containing approximated information for the variables. The widening operator allows us to guarantee the convergence of the abstract fixpoint computation.

Key Words: concurrent constraint paradigm, abstract analysis.

1 Introduction

The Concurrent Constraint Paradigm (*ccp*, [10]) is a simple, logic model which is different from other (concurrent) programming paradigms mainly due to the notion of store-as-constraint that replaces the classical store-as-valuation model. It is based on an underlying constraint system that handles constraints on variables and deals with partial information. Within this family, [6] introduced the *Timed Concurrent Constraint Language* (*tccp*) by adding to the original *ccp*

[*]This work has been supported by the Andalusian Excellence Project P11-TIC7659.

[†]This work has been partially supported by the EU (FEDER) and the Spanish MINECO under grant TIN 2013-45732-C4-1-P (DAMAS) and by Generalitat Valenciana PROMETEOII/2015/013.

model the notion of time and the ability to capture the absence of information. With these features, one can specify behaviors typical of reactive systems such as *timeouts* or *preemption* actions.

It is well-known that modeling and analyzing concurrent systems by hand can be an extremely hard task. Thus, the development of automatic formal methods is essential. The particular characteristics of *ccp* languages make such task even harder, since we have to deal with technical issues due to the infinite computations (natural to reactive systems), use of negative information (particular for *ccp* languages) and non-determinism.

One well established technique to develop semantic-based program analysis is abstract interpretation [5], which relies on the definition of a specific approximated abstract semantics that captures the information needed to perform the analysis. Typically, one defines an over-approximation of the concrete semantics that includes all possible traces of the system, possibly introducing inexistent ones. This allows one to develop (correct) analysis of *universal* properties. It does not allow to analyze *existential* properties, for instance to verify that there exists a suspension trace. In our proposal, we follow such approach starting from the concrete semantics for *tccp* defined in [4]. This semantics addresses (with the minimal amount of information) all thorniest difficulties of *tccp* (i.e., infinite computations, use of negative information and non-determinism). To the best of our knowledge, this is the only bottom-up and condensed semantics which is fully abstract w.r.t. the *full tccp* language. Therefore, such semantics is particularly well-suited as the base to apply abstract interpretation techniques, which take great advantage from a bottom-up and condensed definition. The fully-abstract denotational semantics of [6] captures just finite computations and has a top-down definition thus it is not well-suited for our purposes.

We define a framework of over-approximated abstract semantics parametric w.r.t. an abstract constraint system. This allows us to recycle the work done for developing abstract domains for logic programs (such as groundness analysis). More interestingly, we can also make new analyses for reactive systems such as non-suspension analysis and universal (safety and liveness) properties. Since we need to preserve the notion of time—to be able to express properties of interest like safety or time-depending properties—the abstract semantics domains are not Noetherian (even if we use finite abstract constraint systems). Thus, in order to have an effective approach we use the widening approach of [2, 5] to ensure finiteness of the analysis. Applicability of our approach is illustrated by showing different analyses over our guiding example, a lift/passenger system. More specifically, we show how properties such as *the lift direction and floor are consistently updated* or *the lift/passenger never suspends* can be analyzed.

2 The *tccp* language

The *tccp* language [6] is particularly suitable to specify both reactive and time critical systems. As the other languages of the *ccp* paradigm [10], it is parametric w.r.t. a *cylindric constraint system* which handles the data information of the program in terms of constraints. The computation progresses as the concurrent and asynchronous activity of several agents that can accumulate information in a *store*, or query information from it. A cylindric constraint system[1] is an algebraic structure $\mathbf{C} = \langle \mathcal{C}, \preceq, \otimes, \oplus, \mathit{false}, \mathit{true}, \mathit{Var}, \exists \rangle$ composed of a set of constraints $\mathcal{C}$ such that $(\mathcal{C}, \preceq)$ is a complete algebraic lattice where $\otimes$ is the *lub* operator, $\oplus$ is the *glb* operator and *false* and *true* are respectively the greatest and the least element of $\mathcal{C}$; *Var* is a denumerable set of variables and $\exists$ existentially quantifies variables over constraints. The *entailment* $\vdash$ is the inverse of $\preceq$.

Given a cylindric constraint system $\mathbf{C}$ and a set of process symbols Π, the syntax of agents is given by the grammar $A ::= \mathsf{skip} \mid \mathsf{tell}\,(c) \mid A \parallel A \mid \exists x\, A \mid \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A \mid \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ A \mid p(\vec{x})$ where c, $c_1, \ldots, c_n$ are finite constraints in $\mathbf{C}$; $p_{/m} \in \Pi$, and $\vec{x}$ denotes a generic tuple of m variables. A *tccp* program is an object of the form $D\,.\,A$, where A is an agent, called *initial agent*, and D is a set of *process declarations* of the form $p(\vec{x}) \mathrel{:-} A$ (for some agent A). The notion of time is introduced by defining a discrete and global clock.

The *operational semantics* of *tccp*, defined in [6], is formally described by a transition system $T = (\mathit{Conf}, \longrightarrow)$. Informally, the $\mathsf{tell}\,(c)$ agent adds the constraint c to the store in the next time instant and then stops. The choice agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ consults the store and non-deterministically executes (at the following time instant) one of the agents A_i whose corresponding guard c_i is entailed by the current store; otherwise, if no guard is entailed by the store, the agent suspends. The conditional agent $\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$ behaves in the current time instant like A (resp. B) if c is (resp. is not) entailed by the store. $A \parallel B$ models the parallel composition of A and B in terms of maximal parallelism. The agent $\exists x\, A$ makes 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 $p(\vec{x}) \mathrel{:-} A$ and then executes A at the following time instant.

Example 2.1

The following code shows a possible *tccp* implementation of a simple lift/passenger system. We assume that the lift is located at a building with $N + 1$ floors numbered as $0, 1, \cdots, N$. The lift process uses variables *CF* and *Dir* to store the current floor where the lift is placed and the movement direction (*up*/*down*), respectively. At each time instant, the lift moves, if possible, to the following floor, according to the current movement direction. When the lift reaches floors 0 or N, then it changes the movement direction. Process *pssngr* models the behavior of a client that wants to take the lift to go from origin floor O to destination floor D. This process makes use of variable *St* to store its state: *wait*, when it is waiting for the lift, *in*, when it is inside the lift and *out*, when the passenger has arrived at the destination floor. We use a simple constraint system composed of the atoms $\{\mathit{up}, \mathit{down}, \mathit{in}, \mathit{out}, \mathit{wait}\}$ and with arithmetic operations over the numbers $\{0, \ldots, N\}$. Due to the monotonicity of the store, streams (written in a list-fashion way) are used to model *imperative-style* variables [6].

$$\mathit{main}(N, O, D) :- \exists\, \mathit{CF}, \mathit{Dir}, \mathit{St}\, \big(\mathit{lift}(N, \mathit{CF}, \mathit{Dir}) \parallel \mathit{pssngr}(\mathit{CF}, O, D, \mathit{St}) \parallel$$
$$\mathsf{tell}\,(\mathit{CF} = [0 \mid \_]) \parallel \mathsf{tell}\,(\mathit{Dir} = [\mathit{up} \mid \_]) \parallel \mathsf{tell}\,(\mathit{St} = [\mathit{wait} | \_])\,\big)$$

$$\mathit{lift}(N, \mathit{CF}, \mathit{Dir}) :- \exists\, \mathit{CF}^{l}, \mathit{Dir}^{l}, F\, \big(\,\mathsf{now}(\mathit{Dir} = [\mathit{up} \mid \_] \wedge \mathit{CF} = [N \mid \_])$$

[1]See [6, 10] for more details on cylindric constraint systems, where traditionally, the *glb* $\oplus$ is not explicitly defined.

```
        then (tell $\left(Dir = [up \mid Dir^l]\right)$ ∥ tell $\left(Dir^l = [down \mid \_]\right)$ ∥ $lift(N, CF, Dir^l)$)
    else now ($Dir = [up \mid \_]$) then (tell $\left(CF = [F \mid CF^l]\right)$ ∥
        ask($true$) → (tell $\left(CF^l = [F+1 \mid \_]\right)$ ∥ $lift(N, CF^l, Dir)$)))
    else now ($Dir = [down \mid \_] \wedge CF = [0 \mid \_]$)
        then (tell $\left(Dir = [down \mid Dir^l]\right)$ ∥ tell $\left(Dir^l = [up \mid \_]\right)$ ∥ $lift(N, CF, Dir^l)$)
    else (tell $\left(CF = [F \mid CF^l]\right)$ ∥
            ask($true$) → (tell $\left(CF^l = [F-1 \mid \_]\right)$ ∥ $lift(N, CF^l, Dir)$))))

$pssngr(CF, O, D, St) :- \exists St'$ (
  ask($CF = [D \mid \_] \wedge St = [in|\_]$) → (tell $\left(St = [in \mid St']\right)$ ∥ tell $\left(St' = [out \mid \_]\right)$)
  + ask($CF = [O \mid \_] \wedge St = [wait|\_]$) → (tell $\left(St = [wait|St']\right)$ ∥ tell $\left(St' = [in \mid \_]\right)$ ∥
                                                   tell $\left(CF = [\_ \mid CF']\right)$ ∥ $pssngr(CF', O, D, St')$)
  + ask(($CF \neq [O \mid \_] \wedge CF \neq [D \mid \_]) \vee (CF = [D \mid \_] \wedge St \neq [in|\_]$)
      $\vee (CF = [O \mid \_] \wedge St \neq [wait|\_])$) → (tell $\left(CF = [\_ \mid CF']\right)$ ∥ $pssngr(CF', O, D, St')$))
```

2.1 The concrete denotational semantics

In this section, we briefly recall the concrete denotational domain and semantics of [4], which is fully-abstract (correct and complete) w.r.t. the small-step operational behavior of *tccp*. The denotational semantics of a *tccp* program [4] consists of a set of *conditional (timed) traces* that represent, in a compact way, all the possible behaviors that the program can manifest when fed with an *input* (initial store). Conditional traces can be seen as hypothetical computations in which, for each time instant, we have a condition representing the information that the global store has to satisfy in order to proceed to the next time instant.

Briefly, a conditional trace is a (possibly infinite) sequence $t_1 \cdots t_n \cdots$ of *conditional states*, which can be of three forms:

conditional store: a pair $\eta \rightarrowtail c$, where η is a *condition* and $c \in \mathbf{C}$ a store;

stuttering: the construct $stutt(C)$, with $C \subseteq \mathbf{C} \setminus \{true\}$;

end of a process: the construct $\boxtimes$.

Intuitively, the conditional store $\eta \rightarrowtail c$ means that, provided condition η is satisfied by the current store, the computation proceeds so that in the following time instant, the store is c. A *condition* η is a pair $\eta = (\eta^+, \eta^-)$ where $\eta^+ \in \mathbf{C}$ and $\eta^- \in \wp(\mathbf{C})$ are called positive and negative condition, respectively. The positive/negative condition represents information that a given store must/must not entail, thus they have to be consistent in the sense that $\forall c^- \in \eta^- \; \eta^+ \nvdash c^-$. The stuttering construct models the suspension of the computation when none of the guards in a non-deterministic agent is satisfied. C is the set of guards in the non-deterministic agent.

Conditional traces are monotone (i.e., for each $t_i = \eta_i \rightarrowtail c_i$ and $t_j = \eta_j \rightarrowtail c_j$ such that $j \geq i$, $c_j \vdash c_i$) and consistent (i.e., each store in a trace does not entail the negative conditions of the following conditional state). $\mathbb{T}$ is the set of all

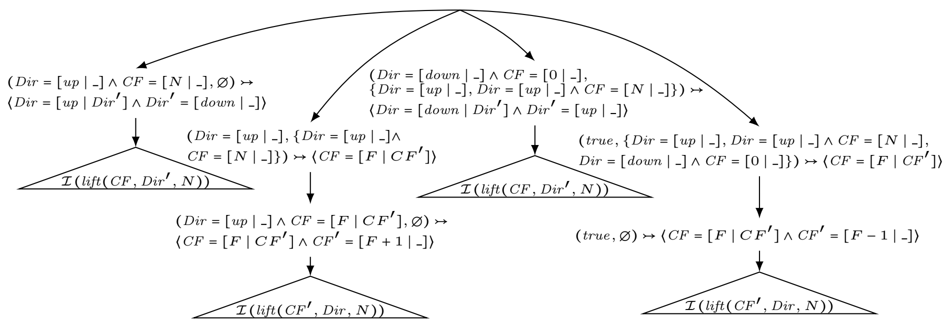

$(Dir = [up \mid \_] \wedge CF = [N \mid \_], \varnothing) \rightarrowtail$
$\langle Dir = [up \mid Dir'] \wedge Dir' = [down \mid \_]\rangle$
$\mathcal{I}(\mathit{lift}(CF, Dir', N))$
$(Dir = [up \mid \_], \{Dir = [up \mid \_] \wedge$
$CF = [N \mid \_]\}) \rightarrowtail \langle CF = [F \mid CF']\rangle$
$(Dir = [up \mid \_] \wedge CF = [F \mid CF'], \varnothing) \rightarrowtail$
$\langle CF = [F \mid CF'] \wedge CF' = [F+1 \mid \_]\rangle$
$\mathcal{I}(\mathit{lift}(CF', Dir, N))$
$(Dir = [down \mid \_] \wedge CF = [0 \mid \_],$
$\{Dir = [up \mid \_], Dir = [up \mid \_] \wedge CF = [N \mid \_]\}) \rightarrowtail$
$\langle Dir = [down \mid Dir'] \wedge Dir' = [up \mid \_]\rangle$
$\mathcal{I}(\mathit{lift}(CF, Dir', N))$
$(true, \{Dir = [up \mid \_], Dir = [up \mid \_] \wedge CF = [N \mid \_],$
$Dir = [down \mid \_] \wedge CF = [0 \mid \_]\}) \rightarrowtail \langle CF = [F \mid CF']\rangle$
$(true, \varnothing) \rightarrowtail \langle CF = [F \mid CF'] \wedge CF' = [F-1 \mid \_]\rangle$
$\mathcal{I}(\mathit{lift}(CF', Dir, N))$

Figure 1: Graph representation of the semantics of the *lift* process.

maximal conditional traces, i.e., infinite traces or finite traces ending with ⊠. With $C \mathrel{\underline{\supset}} C' \iff \forall c \in C.\, \exists c' \in C'.\, c \vdash c'$ we order ⊤ as:

$$(\eta_1^+, \eta_1^-) \rightarrowtail c \cdot r_1 \sqsubseteq (\eta_2^+, \eta_2^-) \rightarrowtail c \cdot r_2 \iff \eta_1^+ \vdash \eta_2^+ \wedge \eta_2^- \mathrel{\underline{\supset}} \eta_1^- \wedge r_1 \sqsubseteq r_2$$
$$\mathit{stutt}\,(\eta_1^-) \cdot r_1 \sqsubseteq \mathit{stutt}\,(\eta_2^-) \cdot r_2 \iff \eta_2^- \mathrel{\underline{\supset}} \eta_1^- \wedge r_1 \sqsubseteq r_2$$

Intuitively, a trace r is smaller than another trace r' iff the conditions of r are more (or equally) restrictive than those of r'. We denote the domain of *maximal conditional trace sets* as $\mathbb{C}$. $(\mathbb{C}, \sqsubseteq, \bigsqcup, \bigsqcap, \top, \bot)$ is a complete lattice, where $M_1 \sqsubseteq M_2 \Leftrightarrow \forall r_1 \in M_1\, \exists r_2 \in M_2.\, r_1 \sqsubseteq r_2$.

The concrete denotational semantics is based on a semantics evaluation function $\mathcal{A}[\![A]\!]_{\mathcal{I}}$ which, given an agent A and an interpretation $\mathcal{I}$, builds the conditional traces associated to A (defined in [4]). The interpretation $\mathcal{I}$ is a function which associates to each process symbol a set of maximal conditional traces "modulo variance".

Definition 2.2 (Interpretations) *Let* $\mathbb{PC} := \{p(\vec{x}) \mid \vec{x}$ *are distinct variables and* p *is a process symbol*$\}$. *An* interpretation *is a function* $\mathcal{I}\colon \mathbb{PC} \to \mathbb{C}$ *modulo variance*[2]. *The semantic domain* $\mathbb{I}$ *is the set of all interpretations ordered by the pointwise extension of* $\sqsubseteq$.

The semantics for a set of process declarations D is the fixpoint $\mathcal{F}[\![D]\!] := \mathit{lfp}(\mathcal{D}[\![D]\!])$ of the continuous operator $\mathcal{D}[\![D]\!]_{\mathcal{I}}(p(\vec{x})) := \bigsqcup_{p(\vec{x}) :- A \in D} \mathcal{A}[\![A]\!]_{\mathcal{I}}$. Proof of full abstraction w.r.t. the operational behavior of *tccp* is given in [4].

Example 2.3 (Semantics of our guiding example)
Consider the lift process defined in Example 2.1. We show in Figure 1 its concrete semantics. Each branch of the tree corresponds to one of the branches of the nested now agents. The first branch (left-to-right order) represents the case in which the direction of the lift is *up* and the current floor is the last one (N). The second branch is taken when the direction is *up* but the current floor is not N (see the negative condition). In that case, the current floor changes from F to $F+1$. The third branch represents the case when the direction of the lift is *down* and the current floor is 0, thus the direction is changed to *up* by adding the constraint $Dir' = [up \mid \_]$. Finally, the fourth branch is taken when all the guards are

[2]Two functions $I, J\colon \mathbb{PC} \to \mathbb{C}$ are *variants*, denoted by $I \cong J$, if for each $\pi \in \mathbb{PC}$ there exists a variable renaming ρ such that $(I(\pi))\rho = J(\pi\rho)$.

not entailed (see the negative condition, composed by all the guards of the nested now agents). In that case, the lift moves to the lower floor $F-1$. In all the aforementioned cases, a recursive call is invoked appropriately. These calls are represented in Figure 1 by the triangles labeled with the interpretation of the process *lift*.

3 The (finite) abstract semantics for *tccp*

In this section, we define our over-approximated abstract semantics for *tccp*. Our abstract semantics is parametric w.r.t. an approximation of the underlying constraint system.

The problem of abstracting constraint systems in the *ccp* paradigm was studied in [7, 11], where abstraction meant loss of completeness but not of correctness. However, for the *tccp* case, due to the strong synchronization of parallel processes, over-approximation of stores could lead to lose correctness [1].

In our semantic domain, constraints are used in three components: in the positive part of the condition, in the negative part and in the store. Since these three components represent different information of a trace, we need to approximate them differently. Similarly to [1, 3], we use both an over- and an under-approximation of the constraint system. The intuitive idea is that we approximate positive information (positive condition and store) with the over-approximation, whereas we approximate negative information with the under-approximation. This allows us to guarantee that we do not loose concrete behaviors when we abstract the semantics, i.e., it is ensured completeness of the abstract semantics. The over-approximating function $\tau^{+}: \mathcal{C} \to \hat{\mathcal{C}}$ abstracts the constraint system $\mathcal{C}$ into an abstract one $\hat{\mathbf{C}} = \langle \hat{\mathcal{C}}, \hat{\preceq}, \hat{\otimes}, \hat{\oplus}, \widehat{false}, \widehat{true}, Var, \hat{\exists} \rangle$ where $\widehat{true}$ and $\widehat{false}$ are the smallest and the greatest abstract constraint, respectively. We often use the inverse relation $\hat{\vdash}$ of $\hat{\preceq}$. The under-approximating function $\tau^{-}: \wp(\mathcal{C}) \to \check{\mathcal{C}}$ abstracts the constraint system into another abstract constraint system $\check{\mathbf{C}} = \langle \check{\mathcal{C}}, \check{\preceq}, \check{\otimes}, \check{\oplus}, \check{false}, \check{true}, Var, \check{\exists} \rangle$. 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 constraint with a concrete constraint (coming from the program).

Over and under-abstract constraints must satisfy the following properties.

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

Moreover, they must be consistent, which means that the "bridge" relation $\tilde{\vdash} \in \hat{\mathcal{C}} \times \check{\mathcal{C}}$ must hold: $\forall c \in C.\ a \nvdash c \Longrightarrow \tau^{+}(a) \mathrel{\tilde{\nvdash}} \tau^{-}(C)$.

Example 3.1 (Sign abstraction)

Given the standard constraint system with inequalities over integer numbers, we abstract it to the abstract constraint system that contains only information about the sign of the system variables. We define the "positive" abstract constraint system as $\hat{\mathbf{S}} := \langle \mathcal{S}, \Leftarrow, \wedge, \vee, \mathit{false}, \mathit{true}, Var, \exists \rangle$ where $\mathcal{S}$ is the set of finite conjunctions of $\{\mathrm{pos}_x, \mathrm{neg}_x, \mathrm{zero}_x \mid x \in Var\} \cup \{\mathit{false},\ \mathit{true}\}$.

The abstract approximation τ^+ is defined by cases as follows:

$$\tau^+(\mathit{true}) = \mathit{true} \qquad \tau^+(x \le a) = \begin{cases} \mathrm{neg}_x & \text{if } a \le 0 \\ \mathit{true} & \text{if } a > 0 \end{cases} \qquad \tau^+(x \ge a) = \begin{cases} \mathrm{pos}_x & \text{if } a \ge 0 \\ \mathit{true} & \text{if } a < 0 \end{cases}$$

$$\tau^+(\mathit{false}) = \mathit{false} \qquad \tau^+(x = a) = \begin{cases} \mathrm{pos}_x & \text{if } a > 0 \\ \mathrm{neg}_x & \text{if } a < 0 \\ \mathrm{zero}_x & \text{if } a = 0 \end{cases}$$

Dually, we define $\check{\mathbf{S}} := \langle \mathcal{S}, \Rightarrow, \vee, \wedge, \mathit{true}, \mathit{false}, \mathit{Var}, \exists \rangle$, the "negative" abstract constraint system. The τ^- function is defined as $\tau^-(C) := \bigwedge_{c \in C} \tau'(c)$, where

$$\tau'(\mathit{true}) = \mathit{true} \qquad \tau'(x \le a) = \begin{cases} \mathrm{neg}_x & \text{if } a \le 0 \\ \mathit{false} & \text{if } a > 0 \end{cases} \qquad \tau'(x \ge a) = \begin{cases} \mathrm{pos}_x & \text{if } a \ge 0 \\ \mathit{false} & \text{if } a < 0 \end{cases}$$

$$\tau'(\mathit{false}) = \mathit{false} \qquad \tau'(x = a) = \begin{cases} \mathrm{pos}_x & \text{if } a > 0 \\ \mathrm{neg}_x & \text{if } a < 0 \\ \mathrm{zero}_x & \text{if } a = 0 \end{cases}$$

The abstract denotational model $\mathbb{A}$ is formed by *abstract conditional traces*, which are conditional traces where conditions and stores are formed by approximated constraints. An abstract conditional trace is said to be *valid* when all its abstract conditions are consistent. An abstract condition (c^+, c^-) is not consistent when $\tau^+(c^+) \tilde{\vdash} \tau^-(c^-)$.

It is worth noting that $(\mathbb{A}, \sqsubseteq, \bigsqcup, \bigsqcap, \mathbf{A}, \bot)$ is a complete lattice.

3.1 The abstract semantics

Now we are ready to define our abstraction approach which works in two steps: the first one abstracts information, and the second one *folds* suspending traces. Formally, concrete and abstract domains are related by the following functions: $(\mathbb{C}, \sqsubseteq) \underset{\alpha^{\mathcal{C}}}{\overset{\gamma^{\mathcal{C}}}{\leftrightarrows}} (\mathbb{A}, \sqsubseteq) \underset{\mathit{fold}}{\overset{\mathit{unfold}}{\leftrightarrows}} (\mathbb{A}, \sqsubseteq)$.

The abstraction function $\alpha^{\mathcal{C}}$ applies τ^+ to each positive condition and store and τ^- to each negative condition occurring in the considered trace. $\alpha^{\mathcal{C}}$ is parametric to the abstraction of the constraint system. The adjoint of $\alpha^{\mathcal{C}}$ is the concretization function $\gamma^{\mathcal{C}}$ that, given a set of abstract traces, produces all the concrete traces that can be approximated with it. For example, given a trace of the form $r = \mathit{stutt}(\{X > 5\}) \cdot (X > 6, \{Y < 0\}) \rightarrowtail X > 9$ the sign abstraction results in the abstract trace $\alpha^{\mathcal{C}}(r) = \mathit{stutt}(\{\mathrm{pos}_X\}) \cdot (\mathrm{pos}_X, \{\mathrm{neg}_Y\}) \rightarrowtail \mathrm{pos}_X$.

The second step of our abstract scheme, the *fold* abstraction, just collapses together all the consecutive identical $\mathit{stutt}(C)$ states. For example, consider $r = \mathit{stutt}(\{\mathrm{pos}_X\}) \cdot \mathit{stutt}(\{\mathrm{pos}_X\}) \cdot \mathit{stutt}(\{\mathrm{pos}_X\})$, then $\mathit{fold}(r) = \mathit{stutt}(\{\mathrm{pos}_X\})$. The adjoint of this abstraction function is the concretization *unfold* which expands each state $\mathit{stutt}(C)$ into a sequence of repetitions of $\mathit{stutt}(C)$ of arbitrary length. Note that this second step does not guarantee termination of analysis since in *tccp* infinite behaviors are not only due to stuttering computations.

Lemma 3.2 *$(\alpha^{\mathcal{C}}, \gamma^{\mathcal{C}})$, $(\mathit{fold}, \mathit{unfold})$ and their composition $\alpha = \alpha^{\mathcal{C}} \circ \mathit{fold}$ and $\gamma = \mathit{unfold} \circ \gamma^{\mathcal{C}}$ are Galois Insertions.*

Proof Sketch.

It is easy to see that both α and γ are monotonic functions, for all $M \in \mathbb{C}$, $(\gamma \circ \alpha)(M) \sqsupseteq M$, and $\alpha \circ \gamma$ is the identity for $\mathbb{A}$.

The Galois insertion defined before can be naturally lifted to the domain of interpretations. We denote as $\mathbb{I}_{\mathbb{A}} := [\mathbb{PC} \to \mathbb{A}]$ the abstract counterpart of $\mathbb{I}$.

The abstract semantics for a *tccp* program is based on the evaluation function for *tccp* agents defined below. In order to improve readability, we have lighten the definition by omitting some technical details. However, we still need some auxiliary operators and properties, intuitively described below. Their formal definitions are similar to those in [4].

Given a trace r and a constraint c, $r{\downarrow_c}$ denotes the propagation of c in the positive conditions occurring in r. The *hiding operator* $\bar{\exists}\colon \mathit{Var} \times \mathbf{A} \to \mathbf{A}$ hides the information regarding a given variable in a trace. It uses $\hat{\exists}$ and $\check{\exists}$ to hide the information in the positive and in the negative conditions and stores. The $\bar{\|}$ operator composes two traces by consistently merging their conditions and stores. A trace r is said to be *self-sufficient* if the first condition is $(\mathit{true}, \varnothing)$ and each store satisfies the successive condition. Moreover, r is *x-self-sufficient* if $\bar{\exists}_{\mathit{Var} \smallsetminus \{x\}}\, r$ is self-sufficient. In other words, for self-sufficient conditional traces, no additional information (from other agents) is needed in order to *complete* the computation.

Definition 3.3 (Abstract Semantics Evaluation Function for Agents) *Given a tccp agent A and an (abstract) interpretation $\mathcal{I}^\alpha \in \mathbb{I}_{\mathbb{A}}$, we define the* semantics evaluation $\mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha} \in \mathbb{A}$ *by structural induction as follows.*

$$\mathcal{A}^\alpha[\![\mathsf{skip}]\!]_{\mathcal{I}^\alpha} := \{\boxtimes\} \tag{3.1}$$

$$\mathcal{A}^\alpha[\![\mathsf{tell}\,(c)]\!]_{\mathcal{I}^\alpha} := \{(\hat{\mathit{true}}, \check{\mathit{false}}) \rightarrowtail \tau^+(c) \cdot \boxtimes\} \tag{3.2}$$

$$\mathcal{A}^\alpha[\![A \parallel B]\!]_{\mathcal{I}^\alpha} := \bigsqcup\{r_A \mathbin{\bar{\|}} r_B \mid r_A \in \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha},\, r_B \in \mathcal{A}^\alpha[\![B]\!]_{\mathcal{I}^\alpha}\} \tag{3.3}$$

$$\mathcal{A}^\alpha[\![\exists x\, A]\!]_{\mathcal{I}^\alpha} := \bigsqcup\{\bar{\exists}_x\, r \mid r \in \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha},\, r \text{ } \mathit{is}\ x\text{-}\mathit{self}\text{-}\mathit{sufficient}\} \tag{3.4}$$

$$\mathcal{A}^\alpha[\![p(\vec{x})]\!]_{\mathcal{I}^\alpha} := \{(\hat{\mathit{true}}, \check{\mathit{false}}) \rightarrowtail \hat{\mathit{true}} \cdot r \mid r \in \mathcal{I}^\alpha(p(\vec{x}))\} \tag{3.5}$$

$$\begin{aligned}\mathcal{A}^\alpha[\![\sum_{i=1}^{n} \mathsf{ask}(c_i) \to A_i]\!]_{\mathcal{I}^\alpha} := {} & M \sqcup \{\mathit{stutt}\,(\tau^-(\{c_1,\dots,c_n\})) \cdot r \mid r \in M\} \sqcup \\ & \{\mathit{stutt}\,(\tau^-(\{c_1,\dots,c_n\}))\}\end{aligned} \tag{3.6}$$

where $M = \bigsqcup\{(\tau^+(c_i), \check{\mathit{false}}) \rightarrowtail \hat{\mathit{true}} \cdot (r{\downarrow_{\tau^+(c_i)}}) \mid 1 \le i \le n,\, r \in \mathcal{A}^\alpha[\![A_i]\!]_{\mathcal{I}^\alpha}\}$

$$\mathcal{A}^\alpha[\![\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B]\!]_{\mathcal{I}^\alpha} :=$$

$$\{(\tau^+(c), \check{\mathit{false}}) \rightarrowtail \hat{\mathit{true}} \cdot \boxtimes \mid \boxtimes \in \mathcal{A}[\![A]\!]_{\mathcal{I}^\alpha}\} \sqcup \tag{3.7a}$$

$$\bigsqcup\{(c \mathbin{\hat{\times}} \hat{\eta}, \check{\eta}) \rightarrowtail \hat{d} \cdot (r{\downarrow_{\tau^+(c)}}) \mid (\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d} \cdot r \in \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha}, c \mathbin{\hat{\times}} \hat{\eta} \mathrel{\tilde{\nvdash}} \check{\eta}\} \sqcup \tag{3.7b}$$

$$\bigsqcup\{(\tau^+(c), \check{\eta}) \rightarrowtail \hat{\mathit{true}} \cdot r{\downarrow_{\tau^+(c)}}) \mid \mathit{stutt}\,(\check{\eta}) \cdot r \in \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha},\, c \mathrel{\tilde{\nvdash}} \check{\eta}\} \sqcup \tag{3.7c}$$

$$\bigsqcup\{(\hat{\mathit{true}}, \tau^-(\{c\})) \rightarrowtail \hat{\mathit{true}} \cdot \boxtimes \mid \boxtimes \in \mathcal{A}^\alpha[\![B]\!]_{\mathcal{I}^\alpha}\} \sqcup \tag{3.7d}$$

$$\bigsqcup\{(\hat{\eta}, \{c\} \mathbin{\check{\times}} \check{\eta}) \rightarrowtail \hat{d} \cdot r \mid (\hat{\eta}, \check{\eta}) \rightarrowtail \hat{d} \cdot r \in \mathcal{A}^\alpha[\![B]\!]_{\mathcal{I}^\alpha},\, \hat{\eta} \mathrel{\hat{\nvdash}} \tau^-(\{c\})\} \sqcup \tag{3.7e}$$

$$\bigsqcup\{(\hat{\mathit{true}}, \{c\} \mathbin{\check{\times}} \check{\eta}) \rightarrowtail \hat{\mathit{true}} \cdot r \mid \mathit{stutt}\,(\check{\eta}) \cdot r \in \mathcal{A}^\alpha[\![B]\!]_{\mathcal{I}^\alpha}\} \tag{3.7f}$$

Note that all the operations regarding the positive part of conditions and the stores are abstracted with the τ^+ abstraction, whereas all the definitions for the negative condition use the τ^- abstraction.

We explain in more detail some significant cases. The semantics of the $\mathsf{tell}\,(c)$ agent (3.1) has a trace with two conditional states: the first one with condition $(\hat{true}, \check{false})$, which is the less demanding condition since c must be added to the store in any case (in the next time instant). Next, the computation terminates with the end-of-process symbol $\boxtimes$. The parallel, hiding and process call cases are defined like in the concrete semantics. The semantics for the non-deterministic choice (3.6) collects, for each guard c_i, a conditional trace of the form $(\tau^+(c_i), \check{false}) \rightarrowtail \hat{true} \cdot (r{\downarrow_{\tau^+(c_i)}})$. This trace requires that $\tau^+(c_i)$ has to be satisfied by the current store. Then, the constraint $\tau^+(c_i)$ is propagated to the conditions in trace r (the continuation of the computation, which belongs to the semantics of A_i). Furthermore, we collect the stuttering traces, which correspond to the case when the computation suspends. These traces are of the form $\mathit{stutt}\,(\tau^-(\{c_1, \dots, c_n\})) \cdot r$ where r is one of the traces above.

The semantics for a set of process declarations D is the fixpoint $\mathcal{F}^\alpha[\![D]\!] \coloneqq \mathit{lfp}\,(\mathcal{D}^\alpha[\![D]\!])$ of the continuous operator $\mathcal{D}^\alpha[\![D]\!]_{\mathcal{I}^\alpha}(p(\vec{x})) \coloneqq \bigsqcup_{p(\vec{x}):-A \in D} \mathcal{A}^\alpha[\![A]\!]_{\mathcal{I}^\alpha}$. It can be shown that $\mathcal{A}^\alpha$ and $\mathcal{D}^\alpha$ are, the optimal abstractions of $\mathcal{A}$ and $\mathcal{D}$, i.e., $\mathcal{A}^\alpha[\![A]\!] = \alpha \circ \mathcal{A}[\![A]\!] \circ \gamma$ and $\mathcal{D}^\alpha[\![D]\!] = \alpha \circ \mathcal{D}[\![D]\!] \circ \gamma$. Hence, abstract interpretation theory ensures that $\mathcal{F}^\alpha[\![D]\!]$ is the best correct approximation of $\mathcal{F}[\![D]\!]$.

3.2 From infinite to finite semantics

Since the domain of abstract conditional traces is not Noetherian (i.e., it admits infinite increasing chains), the abstract least fixpoint does not necessarily converge in finite time. Our solution is to use a *widening operator* [2, 5] that ensures the convergence of the abstract fixpoint in a finite number of steps.

In the following, we use a representation of sets of abstract conditional traces in terms of *conditional graphs*. These graphs are enriched with the information about the process calls, which is necessary to identify the part of the graph corresponding to each iteration of $\mathcal{D}^\alpha[\![D]\!]$ at the moment of applying the widening operator.

Definition 3.4 *A* conditional graph G *is a triple* $(\mathit{Init}, \mathit{Nodes}, \mathit{Edges})$ *where*

- *Init is the set of initial nodes, each one labeled with a (unique) process symbol, denoted by* $\mathit{init}(G)$
- *Nodes is a set of nodes, each one containing a conditional step, and*
- *Edges is a set of edges between nodes that can be of two kinds: either simple edges* $n \to n'$*, or edges of the form* $n \overset{\rho}{\underset{p}{\Longrightarrow}} n'$ *representing a call to process* p *with variable renaming* ρ*. Edges represent the passage of one time unit.*

$\mathbf{G}$ *denotes the set of all conditional graphs. Moreover,* $n \not\to$ *denotes a node* n *that has no outgoing edges.*

We define the function $\mathit{paths}\colon \mathbf{G} \to \mathbf{A}$ which, given a conditional graph, returns the set of all paths of the graph. When an arc of the form $\overset{\rho}{\underset{p}{\Longrightarrow}}$ is traversed, a variant with fresh variables in the co-domain of the renaming ρ is applied to the nodes that follow in the path and the information of the store is propagated to the positive conditions, similarly to what happens when a call is done. The order relation over graphs $\leq$ is defined as $G_1 \leq G_2 \iff \mathit{paths}\,(G_1) \sqsubseteq \mathit{paths}\,(G_2)$. We denote as $(\mathbb{G}, \leq, \bigvee, \bigwedge, \mathbf{G}, \bot_{\mathbb{G}})$ the complete lattice where $\bigvee$ is the least upper bound operator that joins a set of graphs by combining all the sequences that have a prefix in common in the same path, $\bigwedge$ is the greatest lower bound operator that returns the common parts of a set of graphs and $\bot_{\mathbb{G}}$ is the graph composed only of an empty initial node.

The semantics of a *tccp* process $p(\vec{x})$ can be seen as a conditional graph G with the initial node labeled with p and such that $\mathit{paths}\,(G) = \mathcal{F}^{\alpha}[\![D]\!](p(\vec{x}))$. The graph for the process $p(\vec{x})$ is built by linking the initial node of p to the nodes corresponding to the first conditional states of the semantics of an agent A such that $p(\vec{x}) :- A \in D$. The rest of the graph is built following the denotational semantics of Definition 3.3: each conditional state becomes a node in the graph and it is connected to the following one by a simple edge. When a call to a process $q(\vec{y})$ is found and the declaration $q(\vec{z}) :- A'$ is in D, an arrow $\overset{[\vec{z}/\vec{y}]}{\underset{q}{\Longrightarrow}}$ is added, thus linking the current node to the graph labeled with q by using the variable renaming $[\vec{z}/\vec{y}]$.

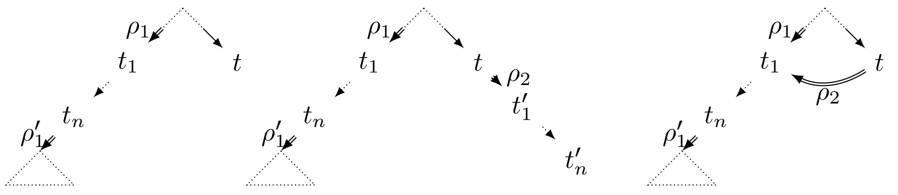
Now we are ready to define our widening operator. Widening operators provide a simple solution to the convergence problem by over-approximating infinite increasing chains in a finite number of steps. A *widening operator* [2, 5] on the lattice $(\mathbb{L}, \leq)$ is a partial function $\triangledown : \mathbb{L} \times \mathbb{L} \to \mathbb{L}$ satisfying: (**covering**) for all $x, y \in \mathbb{L}$ such that $x \leq y$, $x \triangledown y$ exists and $y \leq x \triangledown y$; and (**termination**) for each increasing chain $x_0 \leq x_1 \leq \dots$ the increasing chain defined as $y_0 = x_0$ and $y_{i+1} = y_i \triangledown x_{i+1}$ is not strictly increasing.

We propose a widening operator[3] $\triangledown$ that looks for repeated patterns in consecutive iterations of $\mathcal{D}^{\alpha}[\![D]\!]$ and converges, in a finite number of steps, in a conditional graph that represents an over-approximation of the abstract fixpoint $\mathcal{F}^{\alpha}$. In the sequel, we abuse in notation and write $t \overset{\rho_2}{\underset{p}{\Longrightarrow}} t'_1 \to \dots \to t'_n$ to denote the set of the edges occurring in this path, i.e., $\{t \overset{\rho_2}{\underset{p}{\Longrightarrow}} t'_1, t'_1 \to t'_2 \dots, t'_{n-1} \to t'_n\}$.

Definition 3.5 (Graph widening) *Let $G_1, G_2 \in \mathbb{G}$ such that $G_1 \leq G_2$. The graph widening of G_1 w.r.t. G_2 is defined as $G_1 \triangledown G_2 := G_1 \vee (I, N, E)$ where $I := \mathit{init}(G_2)$, N is the set of nodes that occur in the set of edges E, and*

$$\begin{aligned} E := \bigcup \Big\{ t \overset{\rho_2}{\underset{p}{\Longrightarrow}} t_1 \mid\ & \textit{it exists a subpath in } G_2 \textit{ of the form } t \overset{\rho_2}{\underset{p}{\Longrightarrow}} t'_1 \dots t'_n \not\to \textit{ s.t. an} \\ & \textit{edge} \Longrightarrow \textit{labeled with } p \textit{ does not occur in } t'_1 \dots t'_n \textit{ and it exists a} \\ & \textit{subpath in } G_1 \textit{ of the form} \overset{\rho_1}{\underset{p}{\Longrightarrow}} t_1 \dots t_n \overset{\rho'_1}{\underset{p}{\Longrightarrow}}, \textit{ s.t. an edge} \Longrightarrow \textit{labeled} \\ & \textit{with } p \textit{ does not occur in } t_1 \dots t_n \textit{ and } \forall 1 \leq i \leq n\ \rho_1(t_i) = \rho_2(t'_i) \Big\} \cup \\ \bigcup \Big\{ t \overset{\rho_2}{\underset{p}{\Longrightarrow}} t'_1 \to \dots \to t'_n \mid\ & \textit{it exists a subpath in } G_2 \textit{ on the form } t \overset{\rho_2}{\underset{p}{\Longrightarrow}} t'_1 \dots t'_n \not\to \\ & \textit{s.t. in } t'_1 \dots t'_n \textit{ it does not occur an edge} \Longrightarrow \textit{labeled with } p \textit{ and it does} \\ & \textit{not exist a a subpath in } G_1 \textit{ of the form} \overset{\rho_1}{\underset{p}{\Longrightarrow}} t_1 \dots t_n \overset{\rho'_1}{\underset{p}{\Longrightarrow}}, \textit{ s.t. in } t_1 \dots t_n \textit{ it} \\ & \textit{does not occur an edge} \Longrightarrow \textit{labeled with } p \textit{ and } \forall 1 \leq i \leq n\ \rho_1(t_i) = \rho_2(t'_i) \Big\} \end{aligned}$$

[3]In defining our widening operator, we follow the approach of [2] instead of [5].



(a) G_1 (b) G_2 (c) $G_1 \triangledown G_2$

Figure 2: The graph widening behavior.

At each iteration, the widening checks if a suffix r of a path b in the graph of a process p (which corresponds to the trace produced at the last iteration of p) has already appeared in a previous iteration of p (modulo variables renaming). In this case, it adds an edge, labeled with the necessary variable renaming ρ_2, from the node t precedent to the pattern r to the first node of the equivalent pattern found in the previous widening iteration (first case of Definition 3.5). Otherwise, if no equivalent (modulo variable renaming) pattern is found, the path b is added to the graph (second case of Definition 3.5).

Lemma 3.6 *If the underlying abstract Cylindric Constraint Systems are finite, then the operator $\triangledown$ is a widening operator on $\mathbb{G}$.*

Proof Sketch.

The covering property is a consequence of the fact that the branches of G_2 that are not included by the widening are already present in G_1 modulo variable renaming; that is the reason why a direct edge is added from the last node before the repetition to the equivalent branch detected in G_1.

Termination of the widening is a consequence of the properties of the abstract constraint systems and of the finiteness of the program syntax. By definition, just a finite number of conditional steps can be computed, thus iteration's length is finite. Furthermore, when a repeated pattern is detected, that (possibly cyclic) branch is not further expandable.

Figure 2 shows a graphical representation of the graph widening behavior. To improve readability, in the figure we assume that all process calls involve the same process, thus we just include the renaming for variables in the edges.

Given a *tccp* set of declaration D, we can guarantee ([2]) that the chain

$$I_0 = \bot \qquad I_{i+1} = \begin{cases} I_i & \text{if } \mathcal{D}^\alpha [\![D]\!]_{I_i} \sqsubseteq I_i \\ I_i \triangledown (I_i \sqcup \mathcal{D}^\alpha [\![D]\!]_{I_i}) & \text{otherwise} \end{cases}$$

converges to a graph which is a correct approximation of the abstract semantics in a finite number of steps. That graph contains an initial node for each process declaration such that the subgraph reachable from the initial node represents the corresponding process and subgraphs are linked by edges with renamings.

Example 3.7

Figure 3 shows the conditional graph corresponding to the abstract semantics of the *lift* process. We abstract streams of the concrete Constraint System by posing a depth limit for streams, i.e., we keep the first k values of a stream,

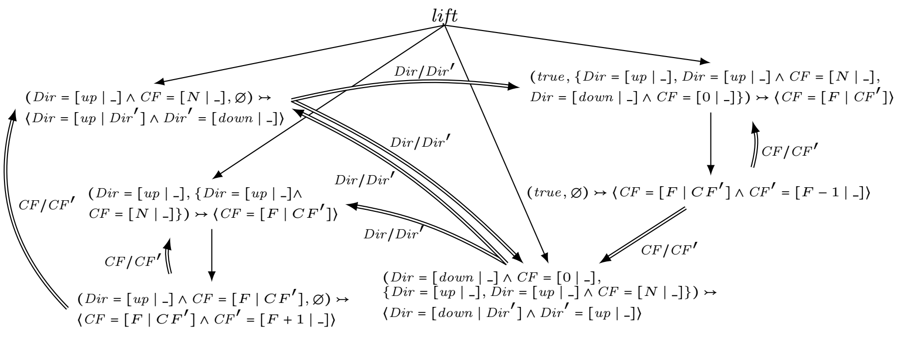

lift
$(Dir = [up \mid \_] \wedge CF = [N \mid \_], \varnothing) \rightarrowtail \langle Dir = [up \mid Dir'] \wedge Dir' = [down \mid \_]\rangle$
$(true, \{Dir = [up \mid \_], Dir = [up \mid \_] \wedge CF = [N \mid \_], Dir = [down \mid \_] \wedge CF = [0 \mid \_]\}) \rightarrowtail \langle CF = [F \mid CF']\rangle$
$(Dir = [up \mid \_], \{Dir = [up \mid \_] \wedge CF = [N \mid \_]\}) \rightarrowtail \langle CF = [F \mid CF']\rangle$
$(true, \varnothing) \rightarrowtail \langle CF = [F \mid CF'] \wedge CF' = [F - 1 \mid \_]\rangle$
$(Dir = [up \mid \_] \wedge CF = [F \mid CF'], \varnothing) \rightarrowtail \langle CF = [F \mid CF'] \wedge CF' = [F + 1 \mid \_]\rangle$
$(Dir = [down \mid \_] \wedge CF = [0 \mid \_], \{Dir = [up \mid \_], Dir = [up \mid \_] \wedge CF = [N \mid \_]\}) \rightarrowtail \langle Dir = [down \mid Dir'] \wedge Dir' = [up \mid \_]\rangle$
Dir/Dir'
Dir/Dir'
Dir/Dir'
Dir/Dir'
CF/CF'
CF/CF'
CF/CF'
CF/CF'

Figure 3: Graph representation of the abstract semantics of the *lift* process.

and then we have the top of the domain. All other constraints are abstracted to themselves. The resulting abstract Constraint System is thus finite.

Due to the application of the widening operator it can be noted how the recursive calls (represented as triangles in Figure 1) are replaced in Figure 3 with the (set of) arcs pointing to the possible continuations of the computation.

---

4 Abstract analysis with an over-approximation

The abstract semantics we have proposed so far is an over-approximation of the concrete semantics. Thus, it allows us to check *universal properties*, i.e., properties that must be satisfied by all the possible behaviors of the system. For instance, it is possible to analyze some temporal properties such as safety (i.e., something bad never happens) or liveness (i.e., something good eventually happens) or to check if a program never suspends.

In order to check whether some invariant property is satisfied by our program, it is necessary to check if every node of the graph respects this property. The properties that can be checked strongly depend on the abstraction of the constraint system. If we want to guarantee that a given abstract constraint c never holds in a computation, we need to check that for every node, either its negative condition contains a store that satisfies c or the positive condition $\hat{\eta}$ is in contradiction with c (i.e., $\hat{\eta} \mathbin{\hat{\otimes}} c = \widehat{false}$). This assures that, for every possible input, c is never produced in the computation.

Similarly, in order to check if an abstract constraint c is always entailed by the current store it is sufficient to check if for each conditional step of the form $(\hat{\eta}, \check{\eta}) \rightarrowtail d$ occurring in the graph, the positive condition merged with the store entails c (i.e., $\hat{\eta} \mathbin{\hat{\otimes}} d \mathrel{\hat{\vdash}} c$). This ensures that for every possible initial constraint d, c is entailed by the store.

Example 4.1

We may be interested in proving several invariant properties on the *lift* process in Example 2.1. For instance, we can try to verify that "the current floor stream *CF* never gets a negative number". To this end, we check all the conditions in the graph in Figure 3, and since we find (at least) a node that does not contradict that *CF* is negative (see the first node of the right branch), we conclude that it cannot be assured that the *lift* process respects this safety property. As a matter of fact, provided we start the computation with an initial state where *CF* is initialized to a negative number, then the last else branch of the program can be taken, and *CF* would keep negative in the subsequent trace.

Consider now the invariant property "each time the direction of the lift is updated, also its floor is updated". In this case, it can be noticed that all the conditional steps in Figure 3 satisfy this property, since whenever the positive condition in the step merged with the store entails that *Dir* has a value, then it is also entailed that *CF* is instantiated.

Verifying liveness properties is harder since it involves analyzing unknown length sequences of steps. For instance, given a process $p(\vec{x})$, assume that we want to check that "every time an abstract constraint c holds, then it exists a future state where another abstract constraint d holds".

Given the conditional graph for $p(\vec{x})$, this property would hold if for each node labeled with a conditional step whose positive condition and store entails c then all paths starting from such node contain a conditional step whose positive condition and store entails d.

Example 4.2

Observe that *lift* process in Example 2.1 satisfies the property "every time the current floor is 0 and the direction is *down*, the direction will be *up* eventually". In fact, the first node of the third branch from the left in Figure 3 is the sole step that contains in its positive condition $CF = [0 \mid \_]$ and $Dir = [down\_]$. Furthermore, for each possible path from this node we find a conditional step where $Dir = [up \mid \_]$ appears in the positive condition or in the store.

Other interesting liveness property that can be analyzed on the *lift* process is "whenever the current floor is 0 it exists a future state when this value changes", i.e., we do not stay indefinitely in floor 0.

Since the number of nodes in the graph is finite, the aforementioned analysis terminates in a finite number of steps.

Let us now analyze non-suspension. Non-suspension analysis consists in assuring that no execution of a *tccp* program suspends. In conditional graphs, in order to check whether $p(\vec{x})$ never suspends, it is sufficient to check that there is no node N in G labeled with a *stutt* construct with an outgoing arc pointing to N itself. Inversely, if the graph contains a stuttering node, we can not guarantee suspension, due to over-approximation of the semantics.

Example 4.3

Consider the semantics of the *lift* process in Figure 3. It is worth noting that the graph does not contain any node labeled with *stutt*. Therefore, we can assure that the *lift* process never suspends.

---

5 Related work

To the best of our knowledge, this is the first attempt to propose an abstract interpretation framework for a concurrent constraint language adhering to the characteristics of *tccp* (negative information, non-determinism and infinite behaviors). In [9], a framework for dataflow analysis of *tcc* and *utcc* programs is presented. The two main differences between these two languages and *tccp* are the notion of time (*tcc* and *utcc* use dedicated timing constructs) and determinism (*vs.* non-determinism of *tccp*). Moreover, in the case studies, [9] uses a $depth(k)$ abstraction to ensure convergence, which consists in a non-selective cut at some point in time. In [8], it was defined a model checking algorithm for *tccp* which allowed us to verify timed-depending properties. Their algorithm was based on the exploration of a graph representation of the program behavior which resembles the graph representation of the semantics defined in this paper. Thus we could as well employ our graph representation to perform (an efficient) model checking. Note however that the abstract semantics that we propose now is not limited to the verification of temporal properties.

Finally, [1] proposes an abstract semantic framework for *tccp* that, differently from our approach, was based on source-to-source transformations. The two approaches are completely different: [1] aimed at using the concrete semantics to execute the transformed (abstract) program. This could be done thanks to a non-trivial transformation of the program (an analysis on the structure of the program was necessary as a preprocess of the transformation). Our approach aims at defining an abstract semantics that, thanks to the characteristics of the concrete denotational semantics, is guaranteed to be correct and we argue that is precise enough to allow the definition of interesting analyses.

6 Conclusions and future work

We have proposed an abstract semantics that, together with a widening operator, is suitable for the definition of different analyses for *full tccp* programs. This is a difficult task because of the presence of infinite computations, use of negative information and non-determinism. However, it is essential since these are the features that make *tccp* well-suited to model reactive systems.

The abstract semantics is an over-approximation, which makes possible to define analysis tools for universal properties. To the best of our knowledge, this is the first proposal that defines an analysis which adaptively ensures termination depending on the program (by means of widening). This should give better results than the non-selective approaches.

This is a first step towards our final goal of defining a rich abstract semantic framework for the analysis of *tccp* programs. We plan to implement the abstract semantics so that we can produce some experimental results. We will need also to implement and develop suitable and useful abstractions for the constraint system, corresponding to the analyses to be performed. We are also interested in defining an under-approximated semantics for *tccp*. Under-approximations produce correct semantics, which means that not all the behaviors are captured, but no spurious behaviors are included. These kind of abstractions allow one to analyze *existential* properties, for instance that there exists a suspension trace.

References

[1] 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.

[2] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise Widening Operators for Convex Polyhedra. *Science of Computer Programming*, 58(1-2):28–56, 2005.

[3] M. Comini, L. Titolo, and A. Villanueva. Abstract Diagnosis for Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming*, 11(4-5):487–502, 2011.

[4] M. Comini, L. Titolo, and A. Villanueva. A Condensed Goal-Independent Bottom-Up Fixpoint Modeling the Behavior of `tccp`. Tech. Rep., Universitat Politècnica de València. Available at http://riunet.upv.es/handle/10251/34328, 2013.

[5] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252, 1977. ACM Press.

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

[7] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional Analysis for Concurrent Constraint Programming. In *Proc. of the 8th Annual IEEE Symposium on Logic in Computer Science*, pages 210–221, 1993. IEEE CS Press.

[8] M. Falaschi and A. Villanueva. Automatic verification of timed concurrent constraint programs. *Theory and Practice of Logic Programming*, 6(3):265–300, 2006.

[9] Moreno Falaschi, Carlos Olarte, and Catuscia Palamidessi. Abstract Interpretation of Temporal Concurrent Constraint Programs. *Theory and Practice of Logic Programming*, 15(3):312–357, 2015.

[10] V. A. Saraswat. *Concurrent Constraint Programming*. The MIT Press, 1993.

[11] E. Zaffanella, R. Giacobazzi, and G. Levi. Abstracting Synchronization in Concurrent Constraint Programming. *Journal of Functional and Logic Programming*, 1997(6), 1997.