Abstract Interpretation of Temporal Concurrent Constraint Programs

Moreno Falaschi, Carlos Olarte, Catuscia Palamidessi

Abstract Interpretation of Temporal Concurrent Constraint Programs

MORENO FALASCHI
Dipartimento di Ingegneria dell'Informazione e Scienze Matematiche
Università di Siena, Italy
E-mail: moreno.falaschi@unisi.it

CARLOS OLARTE
Departamento de Electrónica y Ciencias de la Computación
Pontificia Universidad Javeriana-Cali, Colombia
E-mail: carlosolarte@javerianacali.edu.co

CATUSCIA PALAMIDESSI
INRIA and LIX
Ecole Polytechnique, France
E-mail: catuscia@lix.polytechnique.fr

submitted 19 June 2012; revised 15 May 2013; accepted 3 December 2013

Abstract

Timed Concurrent Constraint Programming (`tcc`) is a declarative model for concurrency offering a logic for specifying reactive systems, i.e. systems that continuously interact with the environment. The universal `tcc` formalism (`utcc`) is an extension of `tcc` with the ability to express mobility. Here mobility is understood as communication of private names as typically done for mobile systems and security protocols. In this paper we consider the denotational semantics for `tcc`, and we extend it to a "collecting" semantics for `utcc` based on closure operators over sequences of constraints. Relying on this semantics, we formalize a general framework for data flow analyses of `tcc` and `utcc` programs by abstract interpretation techniques. The concrete and abstract semantics we propose are compositional, thus allowing us to reduce the complexity of data flow analyses. We show that our method is sound and parametric with respect to the abstract domain. Thus, different analyses can be performed by instantiating the framework. We illustrate how it is possible to reuse abstract domains previously defined for logic programming to perform, for instance, a groundness analysis for `tcc` programs. We show the applicability of this analysis in the context of reactive systems. Furthermore, we make also use of the abstract semantics to exhibit a secrecy flaw in a security protocol. We also show how it is possible to make an analysis which may show that `tcc` programs are suspension free. This can be useful for several purposes, such as for optimizing compilation or for debugging.

KEYWORDS: Timed Concurrent Constraint Programming, Process Calculi, Abstract Interpretation, Denotational Semantics, Reactive Systems

1 Introduction

Concurrent Constraint Programming (ccp) (Saraswat et al. 1991; Saraswat 1993) has emerged as a simple but powerful paradigm for concurrency tied to logic that extends and subsumes both concurrent logic programming (Shapiro 1989) and constraint logic programming (Jaffar and Lassez 1987). The ccp model combines the traditional operational view of process calculi with a *declarative* one based upon logic. This combination allows ccp to benefit from the large body of reasoning techniques of both process calculi and logic. In fact, ccp-based calculi have successfully been used in the modeling and verification of several concurrent scenarios such as biological, security, timed, reactive and stochastic systems (Saraswat et al. 1991; Olarte and Valencia 2008b; Nielsen et al. 2002a; Saraswat et al. 1994; Jagadeesan et al. 2005) (see a survey in (Olarte et al. 2013)).

In the ccp model, agents interact by *telling* and *asking* pieces of information (*constraints*) on a shared store of partial information. The type of constraints that agents can tell and ask is parametric in an underlying constraint system. This makes ccp a flexible model able to adapt to different application domains.

The ccp model has been extended to consider the execution of processes along time intervals or time-units. In tccp (de Boer et al. 2000), the notion of time is identified with the time needed to ask and tell information to the store. In this model, the information in the store is carried through the time-units. On the other hand, in Timed ccp (tcc) (Saraswat et al. 1994), stores are not automatically transferred between time-units. This way, computations during a time-unit proceed monotonically but outputs of two different time-units are not supposed to be related to each other. More precisely, computations in tcc take place in bursts of activity at a rate controlled by the environment. In this model, the environment provides a stimulus (input) in the form of a constraint. Then the system, after a finite number of internal reductions, outputs the final store (a constraint) and waits for the next interaction with the environment. This view of *reactive computation* is akin to synchronous languages such as Esterel (Berry and Gonthier 1992) where the system reacts continuously with the environment at a rate controlled by the environment. Hence, these languages allow to program safety critical applications as control systems, for which it is fundamental to provide tools aiming at helping to develop correct, secure, and efficient programs.

Universal tcc (Olarte and Valencia 2008b) (utcc), adds to tcc the expressiveness needed for *mobility*. Here we understand mobility as the ability to communicate private names (or variables) much like in the π-calculus (Milner et al. 1992). Roughly, a tcc *ask* process **when** c **do** P executes the process P only if the constraint c can be entailed from the store. This idea is generalized in utcc by a parametric ask that executes $P[\vec{t}/\vec{x}]$ when the constraint $c[\vec{t}/\vec{x}]$ is entailed from the store. Hence the variables in $\vec{x}$ act as formal parameters of the ask operator. This simple change allowed to widen the spectrum of application of ccp-based languages to scenarios such as verification of security protocols (Olarte and Valencia 2008b) and service oriented computing (López et al. 2009).

Several domains and frameworks (e.g., (Cousot and Cousot 1992; Armstrong et al.

1998; Codish et al. 1999)) have been proposed for the analysis of logic programs. The particular characteristics of timed `ccp` programs pose additional difficulties for the development of such tools in this language. Namely, the concurrent, timed nature of the language, and the synchronization mechanisms based on entailment of constraints (blocking asks). Aiming at statically analyzing `utcc` as well as `tcc` programs, we have to consider the additional technical issues due to the infinite internal computations generated by parametric asks as we shall explain later.

We develop here a *compositional* semantics for `tcc` and `utcc` that allows us to describe the behavior of programs and collects all concrete information needed to properly abstract the properties of interest. This semantics is based on closure operators over sequences of constraints along the lines of (Saraswat et al. 1994). We show that parametric asks in `utcc` of the form $(\mathbf{abs}\ \vec{x}; c)\, P$ can be neatly characterized as closure operators. This characterization is shown to be somehow dual to the semantics for the local operator $(\mathbf{local}\, \vec{x})\, P$ that restricts the variables in $\vec{x}$ to be local to P. We prove the semantics to be fully abstract w.r.t. the operational semantics for a significant fragment of the calculus.

We also propose an abstract semantics which approximates the concrete one. Our framework is formalized by abstract interpretation techniques and is parametric w.r.t. the abstract domain. It allows us to exploit the work done for developing abstract domains for logic programs. Moreover, we can make new analyses for reactive and mobile systems, thus widening the reasoning techniques available for `tcc` and `utcc`, such as type systems (Hildebrandt and López 2009), logical characterizations (Mendler et al. 1995; Nielsen et al. 2002a; Olarte and Valencia 2008b) and semantics (Saraswat et al. 1994; de Boer et al. 1995; Nielsen et al. 2002a).

The abstraction we propose proceeds in two-levels. First, we approximate the constraint system leading to an abstract constraint system. We give the sufficient conditions which have to be satisfied for ensuring the soundness of the abstraction. Next, to obtain efficient analyses, we abstract the infinite sequences of (abstract) constraints obtained from the previous step. Our semantics is then computable and compositional. Thus, it allows us to master the complexity of the data-flow analyses. Moreover, the abstraction *over-approximates* the concrete semantics, thus preserving safety properties.

To the best of our knowledge, this is the first attempt to propose a compositional semantics and an abstract interpretation framework for a language adhering to the above-mentioned characteristics of `utcc`. Hence we can develop analyses for several applications of `utcc` or its sub-calculus `tcc` (see e.g., (Olarte et al. 2013)). In particular, we instantiate our framework in three different scenarios. The first one presents an abstraction of a cryptographic constraint system. We use the abstract semantics to bound the number of messages that a spy may generate, in order to exhibit a secrecy flaw in a security protocol written in `utcc`. The second one tailors an abstract domain for groundness and type dependency analysis in logic programming to perform a groundness analysis of a `tcc` program. This analysis is proven useful to derive a property of a control system specified in `tcc`. Finally, we present an analysis that may show that a `tcc` program is suspension free. This analysis can be used later for optimizing compilation or for debugging purposes.

The ideas of this paper stem mainly from the works of the authors in (de Boer et al. 1995; Falaschi et al. 1997a; Falaschi et al. 1997b; Nielsen et al. 2002a; Olarte and Valencia 2008a) to give semantic characterization of ccp calculi and from the works in (Falaschi et al. 1993; Codish et al. 1994; Falaschi et al. 1997a; Zaffanella et al. 1997; Falaschi et al. 2007) to provide abstract interpretation frameworks to analyze concurrent logic-based languages. A preliminary short version of this paper without proofs was published in (Falaschi et al. 2009). In this paper we give many more examples and explanations. We also refine several technical details and present full proofs. Furthermore, we develop a new application for analyzing suspension-free tcc programs.

The rest of the paper is organized as follows. Section 2 recalls the notion of constraint system and the operational semantics of tcc and utcc. In Section 3 we develop the denotational semantics based on sequences of constraints. Next, in Section 4, we study the abstract interpretation framework for tcc and utcc programs. The three instances and the applications of the framework are presented in Section 5. Section 6 concludes.

2 Preliminaries

Process calculi based on the ccp paradigm are parametric in a *constraint system* specifying the basic constraints agents can tell and ask. These constraints represent a piece of (partial) information upon which processes may act. The constraint system hence provides a signature from which constraints can be built. Furthermore, the constraint system provides an *entailment* relation ($\vdash$) specifying inter-dependencies between constraints. Intuitively, $c \vdash d$ means that the information d can be deduced from the information represented by c. For example, $x > 60 \vdash x > 42$.

Here we consider an abstract definition of constraint systems as cylindric algebras as in (de Boer et al. 1995). The notion of constraint system as first-order formulas (Smolka 1994; Nielsen et al. 2002a; Olarte and Valencia 2008b) can be seen as an instance of this definition. All results of this paper still hold, of course, when more concrete systems are considered.

Definition 1 (*Constraint System*)
A cylindric constraint system is a structure $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \mathtt{t}, \mathtt{f}, \mathit{Var}, \exists, D\rangle$ s.t.
- $\langle \mathcal{C}, \leq, \sqcup, \mathtt{t}, \mathtt{f}\rangle$ is a lattice with $\sqcup$ the *lub* operation (representing the logical *and*), and t, f the least and the greatest elements in $\mathcal{C}$ respectively (representing true and false). Elements in $\mathcal{C}$ are called *constraints* with typical elements $c, c', d, d' ....$ If $c \leq d$ and $d \leq c$ we write $c \cong d$. If $c \leq d$ and $c \ncong d$, we write $c < d$.
- *Var* is a denumerable set of variables and for each $x \in \mathit{Var}$ the function $\exists x : \mathcal{C} \to \mathcal{C}$ is a cylindrification operator satisfying: (1) $\exists x(c) \leq c$. (2) If $c \leq d$ then $\exists x(c) \leq \exists x(d)$. (3) $\exists x(c \sqcup \exists x(d)) \cong \exists x(c) \sqcup \exists x(d)$. (4) $\exists x \exists y(c) \cong \exists y \exists x(c)$. (5) For an increasing chain $c_1 < c_2 < c_3 ...$, $\exists x \bigsqcup_i c_i \cong \bigsqcup_i \exists x(c_i)$.
- For each $x, y \in \mathit{Var}$, the constraint $d_{xy} \in D$ is a *diagonal element* and it satisfies: (1) $d_{xx} \cong \mathtt{t}$. (2) If z is different from x, y then $d_{xy} \cong \exists z(d_{xz} \sqcup d_{zy})$. (3) If x is different from y then $c \leq d_{xy} \sqcup \exists x(c \sqcup d_{xy})$.

The cylindrification operators model a sort of existential quantification, helpful for hiding information. We shall use $fv(c) = \{x \in Var \mid \exists x(c) \not\cong c\}$ to denote the set of free variables that occur in c. If x occurs in c and $x \notin fv(c)$, we say that x is bound in c. We use $bv(c)$ to denote the set of bound variables in c.

Properties (1) to (4) are standard. Property (5) is shown to be required in (de Boer et al. 1995) to establish the semantic adequacy of `ccp` languages when infinite computations are considered. Here, the continuity of the semantic operator in Section 3 relies on the continuity of $\exists$ (see Proposition 2). Below we give some examples on the requirements to satisfy this property in the context of different constraint systems.

The diagonal element d_{xy} can be thought of as the equality $x = y$. Properties (1) to (3) are standard and they allow us to define substitutions of the form $[t/x]$ required, for instance, to represent the substitution of formal and actual parameters in procedure call. We shall give a formal definition of them in Notation 2.

Let us give some examples of constraint systems. The finite domain constraint system (FD) (Hentenryck et al. 1998) assumes variables to range over finite domains and, in addition to equality, one may have predicates that restrict the possible values of a variable to some finite set, for instance $x < 42$.

The Herbrand constraint system $\mathcal{H}$ consists of a first-order language with equality. The entailment relation is the one we expect from equality, for instance, $f(x, y) = f(g(a), z)$ must entail $x = g(a)$ and $y = z$. $\mathcal{H}$ may contain non-compact elements to represent the limit of infinite chains. To see this, let s be the successor constructor, $\exists y(x = s(s^n(y)))$ be denoted as the constraint $\mathtt{gt}(x, n)$ (i.e., $x > n$) and $\{\mathtt{gt}(x, n)\}_n$ be the ascending chain $\mathtt{gt}(x, 0) < \mathtt{gt}(x, 1) < \cdots$. We note that $\exists x(\mathtt{gt}(x, n)) = \mathtt{t}$ for any n and then, $\bigsqcup\{\exists x(\mathtt{gt}(x, n))\}_n = \mathtt{t}$. Property (5) in Definition 1 dictates that $\exists x \bigsqcup\{\mathtt{gt}(x, n)\}_n$ must be equal to $\mathtt{t}$ (i.e., there exists an x which is greater than any n). For that, we need a constraint, e.g., $\mathtt{inf}(x)$ (a non-compact element), to be the limit $\bigsqcup\{\mathtt{gt}(x, n)\}_n$. We know that $\mathtt{inf}(x) \vdash \mathtt{gt}(x, n)$ for any n and then, $\bigsqcup\{\mathtt{gt}(x, n)\}_n = \mathtt{inf}(x)$ and $\exists x(\mathtt{inf}(x)) = \mathtt{t}$ as wanted. A similar phenomenon arises in the definition of constraint system as Scott information systems in (Saraswat et al. 1991). There, constraints are represented as finite subsets of *tokens* (elementary constraints) built from a given set D. The entailment is similar to that in Definition 1 but restricted to compact elements, i.e., a constraint can be entailed only from a finite set of elementary constraints. Moreover, $\exists$ is extended to be a continuous function, thus satisfying Property (5) in Definition 1. Hence, the Herbrand constraint system in (Saraswat et al. 1991) considers also a non-compact element (different from $\mathtt{f}$) to be the limit of the chain $\{\mathtt{gt}(x, n)\}_n$.

Now consider the Kahn constraint system underlying data-flow languages where equality is assumed along with the constant *nil* (the empty list), the predicate $\mathtt{nempty}(x)$ (x is not *nil*), and the functions $\mathtt{first}(x)$ (the first element of x), $\mathtt{rest}(x)$ (x without its first element) and $\mathtt{cons}(x, y)$ (the concatenation of x and y). If we consider the Kahn constraint system in (Saraswat et al. 1991), the constraint c defined as $\{\mathtt{first}(\mathtt{tail}^n(x)) = \mathtt{first}(\mathtt{tail}^n(y)) \mid n \geq 0\}$ does not entail $\{x = y\}$ since the entailment relation is defined only on compact elements. In Definition 1, we are free to decide if c is different or not from $x = y$. If we equate them, the

constraint $x = y$ is not longer a compact element and then, one has to be careful to only use a compact version of "=" in programs (see Definition 2). A similar situation occurs with the Rational Interval Constraint System (Saraswat et al. 1991) and the constraints $\{x \in [0, 1 + 1/n] \mid n \geq 0\}$ and $x \in [0, 1]$.

All in all many different constraint systems satisfy Definition 1. Nevertheless, one has to be careful since the constraint systems might not be the same as what is naively expected due to the presence of non-compact elements.

We conclude this section by setting some notation and conventions about terms, sequences of constraints, substitutions and diagonal elements. We first lift the relation $\leq$ and the cylindrification operator to sequences of constraints.

Notation 1 (*Sequences of Constraints*)
We denote by $\mathcal{C}^\omega$ (resp. $\mathcal{C}^*$) the set of infinite (resp. finite) sequences of constraints with typical elements $w, w', s, s', ...$ We use W, W', S, S' to range over subsets of $\mathcal{C}^\omega$ or $\mathcal{C}^*$. We use c^ω to denote the sequence $c.c.c....$ The length of s is denoted by $|s|$ and the empty sequence by ϵ. The i-th element in s is denoted by $s(i)$. We write $s \leq s'$ iff $|s| \leq |s'|$ and for all $i \in \{1, \ldots, |s|\}$, $s'(i) \vdash s(i)$. If $|s| = |s'|$ and for all $i \in \{1, ..., |s|\}$ it holds $s(i) \cong s'(i)$, we shall write $s \cong s'$. Given a sequence of variables $\vec{x}$, with $\exists\vec{x}(c)$ we mean $\exists x_1 \exists x_2 ... \exists x_n(c)$ and with $\exists\vec{x}(s)$ we mean the pointwise application of the cylindrification operator to the constraints in s.

We shall assume that the diagonal element d_{xy} is interpreted as the equality $x = y$. Furthermore, following (Giacobazzi et al. 1995), we extend the use of d_{xy} to consider terms as in d_{xt}. More precisely,

Convention 1 (*Diagonal elements*)
We assume that the constraint system under consideration contains an equality theory. Then, diagonal elements d_{xy} can be thought of as formulas of the form $x = y$. We shall use indistinguishably both notations. Given a variable x and a term t (i.e., a variable, constant or n-place function of n terms symbol), we shall use d_{xt} to denote the equality $x = t$. Similarly, given a sequence of distinct variables $\vec{x}$ and a sequence of terms $\vec{t}$, if $|\vec{x}| = |\vec{t}| = n$ then $d_{\vec{x}\vec{t}}$ denotes the constraint $\bigsqcup_{1 \leq i \leq n} x_i = t_i$. If $|\vec{x}| = |\vec{t}| = 0$ then $d_{\vec{x}\vec{t}} = \mathtt{t}$. Given a set of diagonal elements E, we shall write $E \Vdash d_{\vec{x}\vec{t}}$ whenever $d_i \vdash d_{\vec{x}\vec{t}}$ for some $d_i \in E$. Otherwise, we write $E \nVdash d_{\vec{x}\vec{t}}$.

Finally, we set the notation for substitutions.

Notation 2 (*Admissible substitutions*)
Let $\vec{x}$ be a sequence of pairwise distinct variables and $\vec{t}$ be a sequence of terms s.t. $|\vec{t}| = |\vec{x}|$. We denote by $c[\vec{t}/\vec{x}]$ the constraint $\exists\vec{x}(c \sqcup d_{\vec{x}\vec{t}})$ which represents abstractly the constraint obtained from c by replacing the variables $\vec{x}$ by $\vec{t}$. We say that $\vec{t}$ is admissible for $\vec{x}$, notation $adm(\vec{x}, \vec{t})$, if the variables in $\vec{t}$ are different from those in $\vec{x}$. If $|\vec{x}| = |\vec{t}| = 0$ then trivially $adm(\vec{x}, \vec{t})$. Similarly, we say that the substitution $[\vec{t}/\vec{x}]$ is admissible iff $adm(\vec{x}, \vec{t})$. Given an admissible substitution $[\vec{t}/\vec{x}]$, from Property (3) of diagonal elements in Definition 1, we note that $c[\vec{t}/\vec{x}] \sqcup d_{\vec{x}\vec{t}} \vdash c$.

2.1 Reactive Systems and Timed CCP

Reactive systems (Berry and Gonthier 1992) are those that react continuously with their environment at a rate controlled by the environment. For example, a controller or a signal-processing system, receives a stimulus (input) from the environment. It computes an output and then, waits for the next interaction with the environment.

In the `ccp` model, the shared store of constraints grows monotonically, i.e., agents cannot drop information (constraints) from it. Then, a system that changes the state of a variable as in "*signal* = *on*" and "*signal* = *off*" leads to an inconsistent store.

Timed `ccp` (`tcc`) (Saraswat et al. 1994) extends `ccp` for reactive systems. Time is conceptually divided into *time intervals* (or *time-units*). In a particular time interval, a `ccp` process P gets an input c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. The resting point determines also a residual process Q which is then executed in the next time-unit. The resulting store d is not automatically transferred to the next time-unit. This way, computations during a time-unit proceed monotonically but outputs of two different time-units are not supposed to be related to each other. Therefore, the variable *signal* in the example above may change its value when passing from one time-unit to the next one.

Definition 2 (`tcc` *Processes*)
The set *Proc* of `tcc` processes is built from the syntax

$$P, Q \;:=\; \mathbf{skip} \;\mid\; \mathbf{tell}(c) \;\mid \mathbf{when}\ c\ \mathbf{do}\ P \;\mid\; P \parallel Q \;\mid\; (\mathbf{local}\,\vec{x})\,P \;\mid\; \mathbf{next}\,P \;\mid\; \mathbf{unless}\ c\ \mathbf{next}\,P \;\mid\; p(\vec{t})$$

where c is a compact element of the underlying constraint system. Let $\mathcal{D}$ be a set of process declarations of the form $p(\vec{x}) :\!- P$. A `tcc` program takes the form $\mathcal{D}.P$. We assume $\mathcal{D}$ to have a unique process definition for every process name, and recursive calls to be guarded by a **next** process.

The process **skip** does nothing thus representing inaction. The process **tell**(c) adds c to the store in the current time interval making it available to the other processes. The process **when** c **do** P *asks* if c can be deduced from the store. If so, it behaves as P. In other case, it remains blocked until the store contains at least as much information as c. The parallel composition of P and Q is denoted by $P \parallel Q$. Given a set of indexes $I = \{1, ..., n\}$, we shall use $\prod_{i \in I} P_i$ to denote the parallel composition $P_1 \parallel ... \parallel P_n$. The process $(\mathbf{local}\,\vec{x})\,P$ *binds* $\vec{x}$ in P by declaring it private to P. It behaves like P, except that all the information on the variables $\vec{x}$ produced by P can only be seen by P and the information on the global variables in $\vec{x}$ produced by other processes cannot be seen by P.

The process $\mathbf{next}\,P$ is a *unit-delay* that executes P in the next time-unit. The *time-out* **unless** c $\mathbf{next}\,P$ is also a unit-delay, but P is executed in the next time-unit if and only if c is not entailed by the final store at the current time interval. We use $\mathbf{next}^n P$ as a shorthand for $\mathbf{next} \ldots \mathbf{next}\,P$, with **next** repeated n times.

We extend the definition of free variables to processes as follows: $fv(\mathbf{skip}) = \emptyset$; $fv(\mathbf{tell}(c)) = fv(c)$; $fv(\mathbf{when}\ c\ \mathbf{do}\ Q) = fv(c) \cup fv(Q)$; $fv(\mathbf{unless}\ c\ \mathbf{next}\ Q) = fv(c) \cup$

$fv(Q)$; $fv(Q \parallel Q') = fv(Q) \cup fv(Q')$; $fv((\mathbf{local}\,\vec{x})\,Q) = fv(Q) \setminus \vec{x}$; $fv(\mathbf{next}\,Q) = fv(Q)$; $fv(p(\vec{t})) = vars(\vec{t})$ where $vars(\vec{t})$ is the set of variables occurring in $\vec{t}$. A variable x is bound in P if x occurs in P and $x \notin fv(P)$. We use $bv(P)$ to denote the set of bound variables in P.

Assume a (recursive) process definition $p(\vec{x}) :\!- P$ where $fv(P) \subseteq \vec{x}$. The call $p(\vec{t})$ reduces to $P[\vec{t}/\vec{x}]$. Recursive calls in P are assumed to be guarded by a **next** process to avoid non-terminating sequences of recursive calls during a time-unit (see (Saraswat et al. 1994; Nielsen et al. 2002a)).

In the forthcoming sections we shall use the idiom $!\,P$ defined as follows:

Notation 3 (*Replication*)
The replication of P, denoted as $!\,P$, is a short hand for a call to a process definition $\mathtt{bang}_P() :\!- P \parallel \mathbf{next}\,\mathtt{bang}_P()$. Hence, $!\,P$ means $P \parallel \mathbf{next}\,P \parallel \mathbf{next}^2 P$....

2.2 Mobile behavior and utcc

As we have shown, interaction of tcc processes is asynchronous as communication takes place through the shared store of partial information. Similar to other formalisms, by defining local (or private) variables, tcc processes specify boundaries in the interface they offer to interact with each other. Once these interfaces are established, there are few mechanisms to modify them. This is not the case e.g., in the π-calculus (Milner et al. 1992) where processes can change their communication patterns by exchanging their private names. The following example illustrates the limitation of *ask* processes to communicate values and local variables.

Example 1
Let $\mathtt{out}(\cdot)$ be a constraint and let $P = \mathbf{when}\ \mathtt{out}(x)\ \mathbf{do}\ R$ be a system that must react when receiving a stimulus (i.e., an input) of the form $\mathtt{out}(n)$ for $n > 0$. We notice that P in a store $\mathtt{out}(42)$ does not execute R since $\mathtt{out}(42) \not\vdash \mathtt{out}(x)$.

The key point in the previous example is that x is a free-variable and hence, it does not act as a formal parameter (or place holder) for every term t such that $\mathtt{out}(t)$ is entailed by the store.

In (Olarte and Valencia 2008b), tcc is extended for *mobile reactive* systems leading to *universal timed* ccp (utcc). To model mobile behavior, utcc replaces the ask operation **when** c **do** P with a parametric ask construction, namely $(\mathbf{abs}\,\vec{x}; c)\,P$. This process can be viewed as a λ-*abstraction* of the process P on the variables $\vec{x}$ under the constraint (or with the *guard*) c. Intuitively, for all admissible substitution $[\vec{t}/\vec{x}]$ s.t. the current store entails $c[\vec{t}/\vec{x}]$, the process $(\mathbf{abs}\,\vec{x}; c)\,P$ performs $P[\vec{t}/\vec{x}]$. For example, $(\mathbf{abs}\,x; \mathtt{out}(x))\,R$ in a store entailing both $\mathtt{out}(z)$ and $\mathtt{out}(42)$ executes $R[42/x]$ and $R[z/x]$.

Definition 3 (utcc *Processes and Programs*)
The utcc processes and programs result from replacing in Definition 2 the expression **when** c **do** P with $(\mathbf{abs}\,\vec{x}; c)\,P$ where the variables in $\vec{x}$ are pairwise distinct.

When $|\vec{x}| = 0$ we write **when** c **do** P instead of (**abs** $\epsilon; c)\, P$. Furthermore, the process (**abs** $\vec{x}; c)\, P$ binds $\vec{x}$ in P and c. We thus extend accordingly the sets $fv(\cdot)$ and $bv(\cdot)$ of free and bound variables.

From a programming point of view, we can see the variables $\vec{x}$ in the abstraction (**abs** $\vec{x}; c)\, P$ as the formal parameters of P. In fact, the `utcc` calculus was introduced in (Olarte and Valencia 2008b) with replication ($!\, P$) and without process definitions since replication and abstractions are enough to encode recursion. Here we add process definitions to properly deal with `tcc` programs with recursion which are more expressive than those without it (see (Nielsen et al. 2002b)) and we omit replication to avoid redundancy in the set of operators (see Notation 3). We thus could have dispensed with the next-guarded restriction in Definition 2 for `utcc` programs. Nevertheless, in order to give a unified presentation of the forthcoming results, we assume that `utcc` programs adhere also to that restriction.

We conclude with an example of mobile behavior where a process P sends a local variable to Q. Then, both processes can communicate through the shared variable.

Example 2 (*Scope extrusion*)
Assume two components P and Q of a system such that P creates a local variable that must be shared with Q. This system can be modeled as

$$P \;=\; (\mathbf{local}\, x)\, (\mathbf{tell}(\mathtt{out}(x)) \parallel P') \qquad Q \;=\; (\mathbf{abs}\ z; \mathtt{out}(z))\, Q'$$

We shall show later that the parallel composition of P and Q evolves to a process of the form $P' \parallel Q'[x/z]$ where P' and Q' share the local variable x created by P. Then, any information produced by P' on x can be seen by Q' and vice versa.

2.3 Operational Semantics (SOS)

We take inspiration on the structural operational semantics (SOS) for *linear* `ccp` in (Fages et al. 2001; Haemmerlé et al. 2007) to define the behavior of processes. We consider *transitions* between *configurations* of the form $\langle \vec{x}; P; c\rangle$ where c is a constraint representing the current store, P a process and $\vec{x}$ is a set of distinct variables representing the bound (local) variables of c and P. We shall use $\gamma, \gamma', \ldots$ to range over configurations. Processes are quotiented by $\equiv$ defined as follows.

Definition 4 (*Structural Congruence*)
Let $\equiv$ be the smallest congruence satisfying: (1) $P \equiv Q$ if they differ only by a renaming of bound variables (alpha-conversion); (2) $P \parallel \mathbf{skip} \equiv P$; (3) $P \parallel Q \equiv Q \parallel P$; and (4) $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$.

The congruence relation $\equiv$ is extended to configurations by decreeing that $\langle \vec{x}; P; c\rangle \equiv \langle \vec{y}; Q; d\rangle$ iff (**local** $\vec{x}$) $P \equiv$ (**local** $\vec{y}$) Q and $\exists \vec{x}(c) \cong \exists \vec{y}(d)$.

Transitions are given by the relations $\longrightarrow$ and $\Longrightarrow$ in Figure 1. The *internal* transition $\langle \vec{x}; P; c\rangle \longrightarrow \langle \vec{x}'; P'; c'\rangle$ should be read as "P with store c reduces, in one internal step, to P' with store c' ". We shall use $\longrightarrow^*$ as the reflexive and transitive closure of $\longrightarrow$. If $\gamma \longrightarrow \gamma'$ and $\gamma' \equiv \gamma''$ we write $\gamma \longrightarrow\equiv \gamma''$. Similarly for $\longrightarrow^*$.

$$\mathrm{R_{TELL}}\ \frac{}{\langle \vec{x}; \mathbf{tell}(c); d\rangle \longrightarrow \langle \vec{x}; \mathbf{skip}; d \sqcup c\rangle} \qquad \mathrm{R_{PAR}}\ \frac{\langle \vec{x}; P; c\rangle \longrightarrow \langle \vec{x} \cup \vec{y}; P'; d\rangle \ , \ \vec{y} \cap fv(Q) = \emptyset}{\langle \vec{x}; P \parallel Q; c\rangle \longrightarrow \langle \vec{x} \cup \vec{y}; P' \parallel Q; d\rangle}$$

$$\mathrm{R_{LOC}}\ \frac{\vec{y} \cap \vec{x} = \emptyset, \vec{y} \cap fv(d) = \emptyset}{\langle \vec{x}; (\mathbf{local}\, \vec{y})\ P; d\rangle \longrightarrow \langle \vec{x} \cup \vec{y}; P; d\rangle}$$

$$\mathrm{R_{ABS}}\ \frac{d \vdash c[\vec{t}/\vec{y}], adm(\vec{y}, \vec{t}),\ \text{and}\ E \not\Vdash d_{\vec{y}\vec{t}}}{\langle \vec{x}; (\mathbf{abs}\ \vec{y}; c; E)\ P; d\rangle \longrightarrow \langle \vec{x}; P[\vec{t}/\vec{y}] \parallel (\mathbf{abs}\ \vec{y}; c; E \cup \{d_{\vec{y}\vec{t}}\})\ P; d\rangle}$$

$$\mathrm{R_{STRVAR}}\ \frac{nf(c) = \exists \vec{x}_1 c_1 \sqcup \cdots \sqcup \exists \vec{x}_n c_n \qquad \vec{y} \cap \vec{x}_i = \emptyset\ \ \text{forall}\ i \in 1..n}{\langle \vec{y}; P; c\rangle \longrightarrow \langle \vec{y} \cup \bigcup \vec{x}_i; P; c_1 \sqcup ... \sqcup c_n\rangle}$$

$$\mathrm{R_{STR}}\ \frac{\langle \vec{x}; Q; c\rangle \longrightarrow \langle \vec{y}; Q'; c''\rangle}{\langle \vec{x}; P; c\rangle \longrightarrow \langle \vec{x}'; P'; c'\rangle} \qquad \text{if } P \equiv Q \text{ and } \langle \vec{x}'; P'; c'\rangle \equiv \langle \vec{y}; Q'; c''\rangle$$

$$\mathrm{R_{CALL}}\ \frac{p(\vec{x}) :\!- P \in \mathcal{D} \qquad adm(\vec{x}, \vec{t})}{\langle \vec{x}; p(\vec{t}); d\rangle \longrightarrow \langle \vec{x}; P[\vec{t}/\vec{x}]; d\rangle} \qquad \mathrm{R_{UNL}}\ \frac{d \vdash c}{\langle \vec{x}; \mathbf{unless}\ c\ \mathbf{next}\ P; d\rangle \longrightarrow \langle \vec{x}; \mathbf{skip}; d\rangle}$$

Observable Transition

$$\mathrm{R_{OBS}}\ \frac{\langle \emptyset; P; c\rangle \longrightarrow^* \langle \vec{x}; Q; d\rangle \not\longrightarrow}{P \xrightarrow{(c, \exists \vec{x}(d))} (\mathbf{local}\, \vec{x})\ F(Q)} \quad \text{where} \quad \mathrm{F(P)} = \begin{cases} F(\mathbf{skip}) = F((\mathbf{abs}\ \vec{x}; c; D)\ Q) = \mathbf{skip} \\ F(P_1 \parallel P_2) = F(P_1) \parallel F(P_2) \\ F(\mathbf{next}\ Q) = F(\mathbf{unless}\ c\ \mathbf{next}\ Q) = Q \end{cases}$$

Fig. 1: SOS. In $\mathrm{R_{STR}}$, $\equiv$ is given in Definition 4. In $\mathrm{R_{ABS}}$ and $\mathrm{R_{CALL}}$, $adm(\vec{x}, \vec{t})$ is defined in Notation 2. In $\mathrm{R_{ABS}}$, E is assumed to be a set of diagonal elements and $\not\Vdash$ is defined in Convention 1. In $\mathrm{R_{STRVAR}}$, $nf(d)$ is defined in Notation 4.

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

The rules in Figure 1 are easily seen to realize the operational intuitions given in Section 2.1. As clarified below, the seemingly missing rule for a **next** process is given by $\mathrm{R_{OBS}}$. Before explaining such rules, let us introduce the following notation needed for $\mathrm{R_{STRVAR}}$.

Notation 4 (Normal Form)
We observe that the store c in a configuration takes the form $\exists \vec{x}_1(d_1) \sqcup ... \sqcup \exists \vec{x}_n(d_n)$ where each $\vec{x}_i$ may be an empty set of variables. The normal form of c, notation $nf(c)$, is the constraint obtained by renaming the variables in c such that for all $i, j \in 1..n$, if $i \neq j$ then the variables in $\vec{x}_i$ do not occur neither bound nor free in d_j. It is easy to see that $c \cong nf(c)$.

- $\mathrm{R_{TELL}}$ says that the process **tell**(c) adds c to the current store d (via the lub operator of the constraint system) and then evolves into **skip**.
- $\mathrm{R_{PAR}}$ says that if P may evolve into P', this reduction also takes place when running in parallel with Q.
- The process $(\mathbf{local}\, \vec{y})\, Q$ adds $\vec{y}$ to the local variables of the configuration and then evolves into Q. The side conditions of the rule $\mathrm{R_{LOC}}$ guarantee that Q runs with a different set of variables from those in the store and those used by other processes.
- We extend the transition relation to consider processes of the form $(\mathbf{abs}\ \vec{y}; c; E)\, Q$ where E is a set of diagonal elements. If E is empty, we write $(\mathbf{abs}\ \vec{y}; c)\, Q$ instead

of $(\mathbf{abs}\ \vec{y}; c; \emptyset)\, Q$. If d entails $c[\vec{t}/\vec{y}]$, then $P[\vec{t}/\vec{y}]$ is executed (Rule $\mathrm{R_{ABS}}$). Moreover, the abstraction persists in the current time interval to allow other potential replacements of $\vec{y}$ in P. Notice that E is augmented with $d_{\vec{y}\vec{t}}$ and the side condition $E \not\Vdash d_{\vec{y}\vec{t}}$ prevents executing $P[\vec{t}/\vec{y}]$ again. The process $P[\vec{t}/\vec{y}]$ is obtained by equating $\vec{y}$ and $\vec{t}$ and then, hiding the information about $\vec{y}$, i.e., $(\mathbf{local}\,\vec{y})\,(!\,\mathbf{tell}(d_{\vec{y}\vec{t}}) \parallel P)$.
- Rule $\mathrm{R_{STRVAR}}$ allows us to *open* the scope of existentially quantified constraints in the store (see Example 3 below). If γ reduces to γ' using this rule then $\gamma \equiv \gamma'$.
- Rule $\mathrm{R_{STR}}$ says that one can use the structural congruence on processes to continue a derivation (e.g., to do alpha conversion). It is worth noticing that we do not allow in this rule to transform the store via the relation $\equiv$ on configurations and then, via $\cong$ on constraints. We shall discuss the reasons behind this choice in Example 3.
-What we observe from $p(\vec{t})$ is $P[\vec{t}/\vec{x}]$ where the formal parameters are substituted by the actual parameter (Rule $\mathrm{R_{CALL}}$).
- Since the process $P = \mathbf{unless}\ c\ \mathbf{next}\, Q$ executes Q in the next time-unit only if the final store at the current time-unit does not entail c, in the rule $\mathrm{R_{UNL}}$ P evolves into **skip** if the current store d entails c.

For the observable transition relation, rule $\mathrm{R_{OBS}}$ says that an observable transition from P labeled with $(c, \exists\vec{x}(d))$ is obtained from a terminating sequence of internal transitions from $\langle\emptyset; P; c\rangle$ to $\langle\vec{x}; Q; d\rangle$. The process to be executed in the next time interval is $(\mathbf{local}\,\vec{x})\,F(Q)$ (the "future" of Q). $F(Q)$ is obtained by removing from Q the **abs** processes that could not be executed and by "unfolding" the sub-terms within **next** and **unless** expressions. Notice that the output of a process hides the local variables ($\exists\vec{x}(d)$) and those variables are also hidden in the next time-unit ($(\mathbf{local}\,\vec{x})\,F(Q)$).

Now we are ready to show that processes in Example 2 evolve into a configuration where a (local) variable can be communicated and shared.

Example 3 (*Scope Extrusion and Structural Rules*)

Let P and Q be as in Example 2. In the following we show the evolution of the process $P \parallel Q$ starting from the store $\exists w(\mathtt{out}(w))$:

$$
\begin{array}{lll}
1 & \langle\emptyset; P \parallel Q; \exists w(\mathtt{out}(w))\rangle & \longrightarrow^* \ \langle\{x\}; \mathbf{tell}(\mathtt{out}(x)) \parallel P' \parallel Q; \exists w(\mathtt{out}(w))\rangle \\
2 & & \longrightarrow^* \ \langle\{x\}; P' \parallel Q; \exists w(\mathtt{out}(w)) \sqcup \mathtt{out}(x)\rangle \\
3 & & \longrightarrow^* \ \langle\{x, w\}; P' \parallel Q; \mathtt{out}(w) \sqcup \mathtt{out}(x)\rangle \\
4 & & \longrightarrow^* \ \langle\{x, w\}; P' \parallel Q_1 \parallel Q'[w/z]; \mathtt{out}(w) \sqcup \mathtt{out}(x)\rangle \\
5 & & \longrightarrow^* \ \langle\{x, w\}; P' \parallel Q_2 \parallel Q'[w/z] \parallel Q'[x/z]; \mathtt{out}(w) \sqcup \mathtt{out}(x)\rangle
\end{array}
$$

where $Q_1 = (\mathbf{abs}\ z; \mathtt{out}(z); \{d_{wz}\})\, Q'$ and $Q_2 = (\mathbf{abs}\ z; \mathtt{out}(z); \{d_{wz}, d_{xz}\})\, Q'$. Observe that P' and $Q'[x/z]$ share the local variable x created by P. The derivation from line 2 to line 3 uses the Rule $\mathrm{R_{STRVAR}}$ to *open* the scope of w in the store $\exists w(\mathtt{out}(w))$. Let $c_1 = \exists w(\mathtt{out}(w)) \sqcup \mathtt{out}(x)$ (store in line 2) and $c_2 = \mathtt{out}(x)$. We know that $c_1 \cong c_2$. As we said before, Rule $\mathrm{R_{STR}}$ allows us to replace structural congruent processes ($\equiv$) but it does not modify the store via the relation $\cong$ on constraints. The reason is that if we replace c_1 in line 2 with c_2, then we will not observe the execution of $Q'[w/x]$.

2.4 Observables and Behavior

In this section we study the input-output behavior of programs and we show that such relation is a function. More precisely, we show that the input-output relation is a (partial) upper closure operator. Then, we characterize the behavior of a process by the sequences of constraints such that the process cannot add any information to them. We shall call this behavior the strongest postcondition. This relation is fundamental to later develop the denotational semantics for `tcc` and `utcc` programs.

Next lemma states some fundamental properties of the internal relation. The proof follows from simple induction on the inference $\gamma \longrightarrow \gamma'$.

Lemma 1 (*Properties of* $\longrightarrow$)
Assume that $\langle \vec{x}; P; c\rangle \longrightarrow \langle \vec{x}'; Q; d\rangle$. Then, $\vec{x} \subseteq \vec{x}'$. Furthermore:
1. (Internal Extensiveness): $\exists \vec{x}'(d) \vdash \exists \vec{x}(c)$, i.e., the store can only be augmented.
2. (Internal Potentiality): If $e \vdash c$ and $d \vdash e$ then $\langle \vec{x}; P; e\rangle \longrightarrow\equiv \langle \vec{x}'; Q; d\rangle$, i.e., a stronger store triggers more internal transitions.
4. (Internal Restartability): $\langle \vec{x}; P; d\rangle \longrightarrow\equiv \langle \vec{x}'; Q; d\rangle$.

2.4.1 Input-Output Behavior

Recall that `tcc` and `utcc` allows for the modeling of reactive systems where processes react according to the stimuli (input) from the environment. We define the behavior of a process P as the relation of its outputs under the influence of a sequence of inputs (constraints) from the environment. Before formalizing this idea, it is worth noticing that unlike `tcc`, some `utcc` processes may exhibit infinitely many internal reductions during a time-unit due to the **abs** operator.

Example 4 (*Infinite Behavior*)
Consider a constant symbol "a", a function symbol f, a unary predicate (constraint) $c(\cdot)$ and let $Q = (\mathbf{abs}\ x; c(x))\,\mathbf{tell}(c(f(x)))$. Operationally, Q in a store $c(a)$ engages in an infinite sequence of internal transitions producing the constraints $c(f(a))$, $c(f(f(a)))$, $c(f(f(f(a))))$ and so on.

The above behavior will arise, for instance, in applications to security as those in Section 5.1. We shall see that the model of the attacker may generate infinitely many messages (constraints) if we do not restrict the length of the messages (i.e., the number of nested applications of f).

Definition 5 (*Input-Output Behavior*)
Let $s = c_1.c_2...c_n$, $s' = c'_1.c'_2...c'_n$ (resp. $w = c_1.c_2...$, $w' = c'_1.c'_2...$) be finite (resp. infinite) sequences of constraints. If $P = P_1 \xrightarrow{(c_1,c'_1)} P_2 \xrightarrow{(c_2,c'_2)} ...P_n \xrightarrow{(c_n,c'_n)} P_{n+1}$ (resp. $P = P_1 \xrightarrow{(c_1,c'_1)} P_2 \xrightarrow{(c_2,c'_2)} ...$) , we write $P \xrightarrow{(s,s')}$ (resp. $P \xrightarrow{(w,w')}_\omega$). We define the *input-output* behavior of P as $io(P) = io^{fin}(P) \cup io^{inf}(P)$ where

$$\begin{aligned} io^{fin}(P) &= \{(s,s') \mid P \xrightarrow{(s,s')}\} \text{ for } s, s' \in \mathcal{C}^* \\ io^{inf}(P) &= \{(w,w') \mid P \xrightarrow{(w,w')}_\omega\} \text{ for } w, w' \in \mathcal{C}^\omega \end{aligned}$$

We recall that the observable transition ($\Longrightarrow$) is defined through a finite number of internal transitions (rule $\mathrm{R_{OBS}}$ in Figure 1). Hence, it may be the case that for some `utcc` processes (e.g., Q in Example 4), $io^{inf} = \emptyset$. For this reason, we distinguish finite and infinite sequences in the input-output behavior relation. We notice that if $w \in io^{inf}(P)$ then any finite prefix of w belongs to $io^{fin}(P)$. We shall call *well-terminated* the processes which do not exhibit infinite internal behavior.

Definition 6 (*Well-termination*)
The process P is said to be *well-terminated* w.r.t. an infinite sequence w if there exists $w' \in \mathcal{C}^\omega$ s.t. $(w, w') \in io^{inf}(P)$.

Note that `tcc` processes are well-terminated since recursive calls must be **next** guarded. The fragment of well-terminated `utcc` processes has been shown to be a meaningful one. For instance, in (Olarte and Valencia 2008a) the authors show that such fragment is enough to encode Turing-powerful formalisms and (López et al. 2009) shows the use of this fragment in the declarative interpretation of languages for structured communications.

We conclude here by showing that the `utcc` calculus is deterministic. The result follows from Lemma 1 (see Appendix A).

Theorem 1 (*Determinism*)
Let s, w and w' be (possibly infinite) sequences of constraints. If both (s, w), $(s, w') \in io(P)$ then $w \cong w'$.

2.4.2 Closure Properties and Strongest Postcondition

The **unless** operator is the only construct in the language that exhibits non-monotonic input-output behavior in the following sense: Let $P = \mathbf{unless}\ c\ \mathbf{next}\, Q$ and $s \leq s'$. If $(s, w), (s', w') \in io(P)$, it may be the case that $w \not\leq w'$. For example, take $Q = \mathbf{tell}(d)$, $s = \mathtt{t}^\omega$ and $s' = c.\,\mathtt{t}^\omega$. The reader can verify that $w = \mathtt{t}\,.d.\,\mathtt{t}^\omega$, $w' = c.\,\mathtt{t}^\omega$ and then, $w \not\leq w'$.

Definition 7 (*Monotonic Processes*)
We say that P is a monotonic process if it does not have occurrences of **unless** processes. Similarly, the program $\mathcal{D}.P$ is monotonic if P and all P_i in a process definition $p_i(\vec{x}) :\!- P_i$ are monotonic.

Now we show that $io(P)$ is a *partial upper closure operator*, i.e., it is a function satisfying *extensiveness* and *idempotence*. Furthermore, if P is *monotonic*, $io(P)$ is a *closure operator* satisfying additionally monotonicity. The proof of this result follows from Lemma 1 (see details in Appendix A).

Lemma 2 (*Closure Properties*)
Let P be a process. Then, $io(P)$ is a function. Furthermore, $io(P)$ is a partial upper closure operator, namely it satisfies:
Extensiveness: If $(s, s') \in io(P)$ then $s \leq s'$.
Idempotence: If $(s, s') \in io(P)$ then $(s', s') \in io(P)$.

Moreover, if P is monotonic, then:
Monotonicity: If $(s_1, s_1') \in io(P)$, $(s_2, s_2') \in io(P)$ and $s_1 \leq s_2$, then $s_1' \leq s_2'$.

A pleasant property of closure operators is that they are uniquely determined by their set of fixpoints, here called the *strongest postcondition*.

Definition 8 (*Strongest Postcondition*)
Given a `utcc` process P, the strongest postcondition of P, denoted by $sp(P)$, is defined as the set $\{s \in \mathcal{C}^\omega \cup \mathcal{C}^* \mid (s,s) \in io(P)\}$.

Intuitively, $s \in sp(P)$ iff P under input s cannot add any information whatsoever, i.e. s is a quiescent sequence for P. We can also think of $sp(P)$ as the set of sequences that P can output under the influence of an arbitrary environment. Therefore, proving whether P satisfies a given property A, in the presence of any environment, reduces to proving whether $sp(P)$ is a subset of the set of sequences (outputs) satisfying the property A. Recall that $io(P) = io^{fin}(P) \cup io^{inf}(P)$. Therefore, the sequences in $sp(P)$ can be finite or infinite.

We conclude here by showing that for the monotonic fragment, the input-output behavior can be retrieved from the strongest postcondition. The proof of this result follows straightforward from Lemma 2 and it can be found in Appendix A.

Theorem 2
Let min be the minimum function w.r.t. the order induced by $\leq$ and P be a monotonic process. Then, $(s,s') \in io(P)$ iff $s' = min(sp(P) \cap \{w \mid s \leq w\})$.

3 A Denotational model for TCC and UTCC

As we explained before, the strongest postcondition relation fully captures the behavior of a process considering any possible output under an arbitrary environment. In this section we develop a denotational model for the strongest postcondition. The semantics is compositional and it is the basis for the abstract interpretation framework that we develop in Section 4.

Our semantics is built on the closure operator semantics for `ccp` and `tcc` in (Saraswat et al. 1991; Saraswat et al. 1994) and (de Boer et al. 1997; Nielsen et al. 2002a). Unlike the denotational semantics for `utcc` in (Olarte and Valencia 2008a), our semantics is more appropriate for the data-flow analysis due to its simpler domain based on sequences of constraints instead of sequences of temporal formulas. In Section 6 we elaborate more on the differences between both semantics.

Roughly speaking, the semantics is based on a continuous immediate consequence operator $T_{\mathcal{D}}$, which computes in a bottom-up fashion the *interpretation* of each process definition $p(\vec{x}) :- P$ in $\mathcal{D}$. Such an interpretation is given in terms of the set of the quiescent sequences for $p(\vec{x})$.

Assume a `utcc` program $\mathcal{D}.P$. We shall denote the set of process names with their formal parameters in $\mathcal{D}$ as *ProcHeads*. We shall call *Interpretations* the set of functions in the domain $\mathit{ProcHeads} \to \mathcal{P}(\mathcal{C}^\omega)$. We shall define the semantics as a function $[\![\cdot]\!]_I : (\mathit{ProcHeads} \to \mathcal{P}(\mathcal{C}^\omega)) \to (\mathit{Proc} \to \mathcal{P}(\mathcal{C}^\omega))$ which given an interpretation I, associates to each process a set of sequences of constraints.

Before defining the semantics, we introduce the following notation.

$$
\begin{array}{lllll}
\mathrm{D}_{\mathrm{SKIP}} & [\![\mathbf{skip}]\!]_I & = & \mathcal{C}^\omega \\
\mathrm{D}_{\mathrm{TELL}} & [\![\mathbf{tell}(c)]\!]_I & = & \uparrow c.\mathcal{C}^\omega \\
\mathrm{D}_{\mathrm{ASK}} & [\![\mathbf{when}\ c\ \mathbf{do}\ P]\!]_I & = & \overline{\uparrow c}.\mathcal{C}^\omega\ \cup\ (\uparrow c.\mathcal{C}^\omega \cap [\![P]\!]_I) \\
\mathrm{D}_{\mathrm{ABS}} & [\![(\mathbf{abs}\ \vec{x}; c)\ P]\!]_I & = & \forall\, \vec{x}([\![\mathbf{when}\ c\ \mathbf{do}\ P]\!]_I) \\
\mathrm{D}_{\mathrm{PAR}} & [\![P \parallel Q]\!]_I & = & [\![P]\!]_I \cap [\![Q]\!]_I \\
\mathrm{D}_{\mathrm{LOC}} & [\![(\mathbf{local}\,\vec{x})\ P]\!]_I & = & \exists\, \vec{x}([\![P]\!]_I) \\
\mathrm{D}_{\mathrm{NEXT}} & [\![\mathbf{next}\ P]\!]_I & = & \mathcal{C}.[\![P]\!]_I \\
\mathrm{D}_{\mathrm{UNL}} & [\![\mathbf{unless}\ c\ \mathbf{next}\ P]\!]_I & = & \overline{\uparrow c}.[\![P]\!]_I \cup\ \uparrow c.\mathcal{C}^\omega \\
\mathrm{D}_{\mathrm{CALL}} & [\![p(\vec{t})]\!]_I & = & I(p(\vec{t}))
\end{array}
$$

Fig. 2: Semantic Equations for `tcc` and `utcc` constructs. Operands ".", $\uparrow$, $\forall$ and $\exists$ are defined in Notation 5. $\overline{A}$ denotes the set complement of A in $\mathcal{C}^\omega$.

Notation 5 (*Closures and Operators on Sequences*)
Given a constraint c, we shall use $\uparrow c$ (the upward closure) to denote the set $\{d \in \mathcal{C} \mid d \vdash c\}$, i.e., the set of constraints entailing c. Similarly, we shall use $\uparrow s$ to denote the set of sequences $\{s' \in \mathcal{C}^\omega \mid s \leq s'\}$. Given $S \subseteq \mathcal{C}^\omega$ and $\mathcal{C}' \subseteq \mathcal{C}$, we shall extend the use of the sequences-concatenation operator "." by declaring that $c.S = \{c.s \mid s \in S\}$, $\mathcal{C}'.s = \{c.s \mid c \in \mathcal{C}'\}$ and $\mathcal{C}'.S = \{c.s \mid c \in \mathcal{C}' \text{ and } s \in S\}$. Furthermore, given a set of sequences of constraints $S \subseteq \mathcal{C}^\omega$, we define:

$$
\begin{array}{lll}
\exists\, \vec{x}(S) & = & \{s \in \mathcal{C}^\omega \mid \text{ there exists } s' \in S \text{ s.t. } \exists \vec{x}(s) \cong \exists \vec{x}(s')\} \\
\forall\, \vec{x}(S) & = & \{\exists \vec{y}(s) \in S \mid \vec{y} \subseteq \mathit{Var}, s \in S \text{ and for all } s' \in \mathcal{C}^\omega, \text{ if } \exists \vec{x}(s) \cong \exists \vec{x}(s'), \\
 & & \qquad d^\omega_{\vec{x}\vec{t}} \leq s' \text{ and } \mathit{adm}(\vec{x}, \vec{t}) \text{ then } s' \in S\}
\end{array}
$$

The operators above are used to define the semantic equations in Figure 2 and explained in the following. Recall that $[\![P]\!]_I$ aims at capturing the strongest post-condition (or quiescent sequences) of P, i.e. those sequences s such that P under input s cannot add any information whatsoever. The process **skip** cannot add any information to any sequence and hence, its denotation is $\mathcal{C}^\omega$ (Equation $\mathrm{D}_{\mathrm{SKIP}}$). The sequences to which **tell**(c) cannot add information are those whose first element entails c, i.e., the upward closure of c (Equation $\mathrm{D}_{\mathrm{TELL}}$). If neither P nor Q can add any information to s, then s is quiescent for $P \parallel Q$. (Equation $\mathrm{D}_{\mathrm{PAR}}$).

We say that s is an $\vec{x}$-variant of s' if $\exists \vec{x}(s) \cong \exists \vec{x}(s')$, i.e., s and s' differ only on the information about $\vec{x}$. Let $S = \exists\, \vec{x}(S')$. We note that $s \in S$ if there is an $\vec{x}$-variant s' of s in S'. Therefore, a sequence s is quiescent for $Q = (\mathbf{local}\,\vec{x})\ P$ if there exists an $\vec{x}$-variant s' of s s.t. s' is quiescent for P. Hence, if P cannot add any information to s' then Q cannot add any information to s (Equation $\mathrm{D}_{\mathrm{LOC}}$).

The process **next** P has no influence on the first element of a sequence. Hence if s is quiescent for P then $c.s$ is quiescent for **next** P for any $c \in \mathcal{C}$ (Equation $\mathrm{D}_{\mathrm{NEXT}}$). Recall that the process $Q =$ **unless** c **next** P executes P in the next time interval if and only if the guard c cannot be deduced from the store in the current time-unit. Then, a sequence $d.s$ is quiescent for Q if either s is quiescent for P or d entails c (Equation $\mathrm{D}_{\mathrm{UNL}}$). This equation can be equivalently written as $\mathcal{C}.[\![P]\!]_I \cup\ \uparrow c.\mathcal{C}^\omega$.

Recall that the interpretation I maps process names to sequences of constraints. Then, the meaning of $p(\vec{t})$ is directly given by the interpretation I (Rule $\mathrm{D}_{\mathrm{CALL}}$).

Let $Q =$ **when** c **do** P. A sequence $d.s$ is quiescent for Q if d does not entail c.

If d entails c, then $d.s$ must be quiescent for P (rule $\mathrm{D_{ASK}}$). In some cases, for the sake of presentation, we may write this equations as:

$$[\![\mathbf{when}\ c\ \mathbf{do}\ P]\!]_I = \{d.s \mid \text{ if } d \vdash c \text{ then } d.s \in [\![P]\!]_I\}$$

Before explaining the Rule $\mathrm{D_{ABS}}$, let us show some properties of $\forall\, \vec{x}(\cdot)$. First, we note that the $\vec{x}$-variables satisfying the condition $d^{\omega}_{\vec{x}\vec{t}} \leq s$ in the definition of $\forall$ are equivalent (see the proof in Appendix B).

Observation 1 (*Equality and $\vec{x}$-variants*)
Let $S \subseteq \mathcal{C}^{\omega}$, $\vec{z} \subseteq Var$ and $s, w \in \mathcal{C}^{\omega}$ be $\vec{x}$-variants such that $d^{\omega}_{\vec{x}\vec{t}} \leq s$, $d^{\omega}_{\vec{x}\vec{t}} \leq w$ and $adm(\vec{x}, \vec{t})$. (1) $s \cong w$. (2) $\exists \vec{z}(s) \in \forall\, \vec{x}(S)$ iff $s \in \forall\, \vec{x}(S)$.

Now we establish the correspondence between the sets $\forall\, \vec{x}([\![P]\!]_I)$ and $[\![P[\vec{t}/\vec{x}]]\!]_I$ which is fundamental to understand the way we defined the operator $\forall$.

Proposition 1
$s \in \forall\, \vec{x}([\![P]\!]_I)$ if and only if $s \in [\![P[\vec{t}/\vec{x}]]\!]_I$ for all admissible substitution $[\vec{t}/\vec{x}]$.

Proof
($\Rightarrow$)Let $s \in \forall\, \vec{x}([\![P]\!]_I)$ and s' be an $\vec{x}$-variant of s s.t. $d^{\omega}_{\vec{x}\vec{t}} \leq s'$ where $adm(\vec{x}, \vec{t})$. By definition of $\forall$, we know that $s' \in [\![P]\!]_I$. Since $d^{\omega}_{\vec{x}\vec{t}} \leq s'$ then $s' \in [\![P]\!]_I \cap \uparrow(d^{\omega}_{\vec{x}\vec{t}})$. Hence $s \in \exists\, \vec{x}([\![P]\!]_I \cap \uparrow(d^{\omega}_{\vec{x}\vec{t}}))$ and we conclude $s \in [\![P[\vec{t}/\vec{x}]]\!]_I$.
($\Leftarrow$) Let $[\vec{t}/\vec{x}]$ be an admissible substitution. Suppose, to obtain a contradiction, that $s \in [\![P[\vec{t}/\vec{x}]]\!]_I$, there exists s' $\vec{x}$-variant of s s.t. $d^{\omega}_{\vec{x}\vec{t}} \leq s'$ and $s' \notin [\![P]\!]_I$ (i.e., $s \notin \forall\, \vec{x}([\![P]\!]_I)$). Since $s \in [\![P[\vec{t}/\vec{x}]]\!]_I$ then $s \in \exists\, \vec{x}([\![P]\!]_I \cap \uparrow d^{\omega}_{\vec{x}\vec{t}})$. Therefore, there exists s'' $\vec{x}$-variant of s s.t. $s'' \in [\![P]\!]_I$ and $d^{\omega}_{\vec{x}\vec{t}} \leq s''$. By Observation 1, $s' \cong s''$ and thus, $s' \in [\![P]\!]_I$, a contradiction. $\square$

A sequence $d.s$ is quiescent for the process $Q = (\mathbf{abs}\ x; c)\, P$ if for all admissible substitution $[\vec{t}/\vec{x}]$, either $d \not\vdash c[\vec{t}/\vec{x}]$ or $d.s$ is also quiescent for $P[\vec{t}/\vec{x}]$, i.e., $d.s \in \forall\, \vec{x}([\![(\mathbf{when}\ c\ \mathbf{do}\ P)]\!]_I)$ (rule $\mathrm{D_{ABS}}$). Notice that we can simply write Equation $\mathrm{D_{ABS}}$ by unfolding the definition of $\mathrm{D_{ASK}}$ as follows:

$$[\![(\mathbf{abs}\ \vec{x}; c)\, P]\!]_I = \forall\, \vec{x}(\overline{\uparrow c}.\mathcal{C}^{\omega}\ \cup\ (\uparrow c.\mathcal{C}^{\omega} \cap [\![P]\!]_I))$$

The reader may wonder why the operator $\forall$ (resp. Rule $\mathrm{D_{ABS}}$) is not entirely dual w.r.t. $\exists$ (resp. Rule $\mathrm{D_{LOC}}$), i.e., why we only consider $\vec{x}$-variants entailing $d_{\vec{x}\vec{t}}$ where $[\vec{t}/\vec{x}]$ is an admissible substitution. To explain this issue, let $Q = (\mathbf{abs}\ x; c)\, P$ where $c = \mathtt{out}\,(x)$ and $P = \mathbf{tell}(\mathtt{out}'(x))$. We know that

$$s = (\mathtt{out}(a) \wedge \mathtt{out}'(a)).\mathtt{t}^{\omega} \in sp(Q)$$

for a given constant a. Suppose that we were to define:

$$[\![Q]\!]_I = \{s \mid \text{for all } x\text{-variant } s' \text{ of } s \text{ if } s'(1) \vdash c \text{ then } s' \in [\![P]\!]_I\}$$

Let $c' = \mathtt{out}(a) \wedge \mathtt{out}'(a) \wedge \mathtt{out}(x)$ and $s' = c'.\mathtt{t}^{\omega}$. Notice that s' is an x-variant of s, $s'(1) \vdash c$ but $s' \notin [\![P]\!]_I$ (since $c' \not\vdash \mathtt{out}'(x)$). Then $s \notin [\![Q]\!]_I$ under this naive definition of $[\![Q]\!]_I$. We thus consider only the $\vec{x}$-variants s' s.t. each element of s' entails $d_{\vec{x}\vec{t}}$. Intuitively, this condition requires that $s'(1) \vdash c \sqcup d_{\vec{x}\vec{t}}$ in Equation $\mathrm{D_{ABS}}$

and hence that $s'(1) \vdash c[\vec{t}/\vec{x}]$. Furthermore $s \in [\![P[\vec{t}/\vec{x}]]\!]_I$ realizes the operational intuition that P runs under the substitution $[\vec{t}/\vec{x}]$. The operational rule $\mathrm{R}_{\mathrm{STRVAR}}$ makes also echo in the design of our semantics: the operator $\forall$ considers constraints of the form $\exists\vec{z}(s)$ where $\vec{z}$ is a (possibly empty) set of variables, thus allowing us to open the existentially quantified constraints as shown in the following example.

Example 5 (*Scope extrusion*)
Let $P =$ **when** $\mathtt{out}(x)$ **do tell**$(\mathtt{out}'(x))$, $Q =$ (**abs** $\vec{x}; \mathtt{out}(x)$) **tell**$(\mathtt{out}'(x))$. We know that $[\![Q]\!]_I = \forall\, x([\![P]\!]_I)$. Assume that $d.s \in [\![P]\!]_I$. Then, d must be in the set:

$$C = \{\exists x(\mathtt{out}(x)), \mathtt{out}(x) \sqcup \mathtt{out}'(x), \exists x(\mathtt{out}(x) \sqcup \mathtt{out}'(x)), \mathtt{out}(y), \mathtt{out}(y) \sqcup \mathtt{out}'(y) \cdots\}$$

where either, $d \nvdash \mathtt{out}(x)$ or $d \vdash \mathtt{out}'(x)$. We note that: (1) $(\exists x(\mathtt{out}(x))).s \notin [\![Q]\!]_I$ since $\mathtt{out}(x) \notin C$. Similarly, $\exists y(\mathtt{out}(y)).s \notin [\![Q]\!]_I$ since $\mathtt{out}(y) \in C$ but the x-variant $\mathtt{out}(x) \sqcup d_{xy} \notin C$ (it does not entail $\mathtt{out}'(x)$). (3) $\mathtt{out}(y).s \notin [\![P]\!]_I$ for the same reason. (4) Let $e = (\mathtt{out}(x) \sqcup \mathtt{out}'(x))$. We note that $e.s \in [\![Q]\!]_I$ since $e \in C$ and there is not an admissible substitution $[t/x]$ s.t. $\exists x(e) \cong \exists x(e[t/x])$. (5) Let $e = (\mathtt{out}(y) \sqcup \mathtt{out}'(y))$. Then, $e.s \in [\![Q]\!]_I$ since $e \in C$ and the x-variant $e \sqcup d_{xy} \in C$. (6) Finally, if $e = \exists x(\mathtt{out}(x) \sqcup \mathtt{out}'(x)).s$, then $e.s \in [\![Q]\!]_I$ as in (4) and (5).

3.1 Compositional Semantics

We choose as semantic domain $\mathbb{E} = (E, \sqsubseteq^c)$ where $E = \{X \mid X \in \mathcal{P}(\mathcal{C}^\omega)$ and $\mathtt{f}^\omega \in X\}$ and $X \sqsubseteq^c Y$ iff $X \supseteq Y$. The bottom of $\mathbb{E}$ is then $\mathcal{C}^\omega$ (the set of all the sequences) and the top element is the singleton $\{\mathtt{f}^\omega\}$ (recall that $\mathtt{f}$ is the greatest element in $(\mathcal{C}, \leq)$). Given two interpretations I_1 and I_2, we write $I_1 \sqsubseteq^c I_2$ iff for all p, $I_1(p) \sqsubseteq^c I_2(p)$.

Definition 9 (*Concrete Semantics*)
Let $[\![\cdot]\!]_I$ be defined as in Figure 2. The semantics of a program $\mathcal{D}.P$ is the least fixpoint of the continuous operator:

$$T_{\mathcal{D}}(I)(p(\vec{t})) = [\![Q[\vec{t}/\vec{x}]]\!]_I \text{ if } p(\vec{x}) :\!- Q \in \mathcal{D}$$

We shall use $[\![P]\!]$ to represent $[\![P]\!]_{lfp(T_{\mathcal{D}})}$.

In the following we prove some fundamental properties of the semantic operator $T_{\mathcal{D}}$, namely, monotonicity and continuity. Before that, we shall show that $\forall$ is a closure operator and it is continuous on the domain $\mathbb{E}$.

Lemma 3 (*Properties of* $\forall$)
$\forall$ is a closure operator, i.e., it satisfies (1) **Extensivity**: $S \sqsubseteq^c \forall\, \vec{x}(S)$; (2) **Idempotency**: $\forall\, \vec{x}(\forall\, \vec{x}(S)) = \forall\, \vec{x}(S)$; and (3) **Monotonicity**: If $S \sqsubseteq^c S'$ then $\forall\, \vec{x}(S) \sqsubseteq^c \forall\, \vec{x}(S')$. Furthermore, (4) $\forall$ is continuous on $(E, \sqsubseteq^c)$.

Proof
The proofs of (1),(2) and (3) are straightforward from the definition of $\forall\, \vec{x}$. The proof of (4) proceeds as follows. Assume a non-empty ascending chain $S_1 \sqsubseteq^c S_2 \sqsubseteq^c S_3 \sqsubseteq^c \ldots$. Lubs in E correspond to set intersection. We shall prove that

$$
\begin{array}{lll}
I_1 & : & p \to \uparrow \mathtt{out}_a(x).\mathcal{C}^\omega \ \cap \ \mathcal{C}.\uparrow \mathtt{out}_a(y).\mathcal{C}^\omega \text{ i.e., } p \to \uparrow \mathtt{out}_a(x).\uparrow \mathtt{out}_a(y).\mathcal{C}^\omega \\
& & q \to \forall\, z(A.\mathcal{C}^\omega) \cap \ \mathcal{C}.I_\perp(q) \text{ i.e., } q \to \forall\, z(A).I_\perp(q) \\
& & r \to \mathcal{C}^\omega \cap \mathcal{C}^\omega = \mathcal{C}^\omega \\
I_2 & : & p \to I_1(p) \\
& & q \to \forall\, z(A.\mathcal{C}^\omega) \cap \ \mathcal{C}.I_1(q) \text{ i.e., } q \to \forall\, z(A).\forall\, z(A.\mathcal{C}^\omega) \cap \ \mathcal{C}.\mathcal{C}.\mathcal{C}^\omega \\
& & r \to I_1(p) \cap I_1(q) \\
\ldots & & \\
I_\omega & : & p \to I_1(p) \\
& & q \to \forall\, z(A).\forall\, z(A).\forall\, z(A)... \\
& & r \to I_\omega(p) \cap I_\omega(q)
\end{array}
$$

Fig. 3: Semantics of the processes in Example 6. $A_1 = \uparrow(\mathtt{out}_a(z) \sqcup \mathtt{out}_b(z))$, $A_2 = \overline{\uparrow \mathtt{out}_a(z)}$ and $A = A_1 \cup A_2$. We abuse of the notation and we write $\forall\, z(A).S$ instead of $\forall\, z(A.\mathcal{C}^\omega) \cap \mathcal{C}.S$.

$\bigcap \forall\, \vec{x}(S_i) = \forall\, \vec{x}(\bigcap S_i)$. The "$\subseteq$" part (i.e., $\sqsupseteq^c$) is trivial since $\forall$ is monotonic. As for the $\bigcap \forall\, \vec{x}(S_i) \subseteq \forall\, \vec{x}(\bigcap S_i)$ part, by extensiveness we know that $\forall\, \vec{x}(S_i) \subseteq S_i$ for all S_i and then, $\bigcap \forall\, \vec{x}(S_i) \subseteq \bigcap S_i$. Let $s \in \bigcap \forall\, \vec{x}(S_i)$. By definition we know that s and all $\vec{x}$-variant s' of s satisfying $d^\omega_{\vec{x}\vec{t}} \leq s'$ for $adm(\vec{x}, \vec{t})$ belong to $\bigcap \forall\, \vec{x}(S_i)$ and then in $\bigcap S_i$. Hence, $s \in \forall\, \vec{x}(\bigcap S_i)$ and we conclude $\bigcap \forall\, \vec{x}(S_i) \subseteq \forall\, \vec{x}(\bigcap S_i)$. $\square$

Proposition 2 (*Monotonicity of $[\![\cdot]\!]$ and continuity of $T_\mathcal{D}$*)
Let P be a process and $I_1 \sqsubseteq^c I_2 \sqsubseteq^c I_3$... be an ascending chain. Then, $[\![P]\!]_{I_i} \sqsubseteq^c [\![P]\!]_{I_{i+1}}$ (**Monotonicity**). Moreover, $[\![P]\!]_{\bigsqcup_{I_i}} = \bigsqcup_{I_i} [\![P]\!]_{I_i}$ (**Continuity**).

Proof
Monotonicity follows easily by induction on the structure of P and it implies the the "$\sqsupseteq^c$" part of continuity. As for the part "$\sqsubseteq^c$" we proceed by induction on the structure of P. The interesting cases are those of the local and the abstraction operator. For $P = (\mathbf{local}\, \vec{x})\, Q$, by inductive hypothesis we know that $[\![Q]\!]_{\bigsqcup_{I_i}} \sqsubseteq^c \bigsqcup_{I_i} [\![Q]\!]_{I_i}$. Since $\exists$ (and therefore $\exists$) is continuous (see Property (5) in Definition 1), we conclude $\exists_{\vec{x}}([\![Q]\!]_{\bigsqcup_{I_i}}) \sqsubseteq^c \bigsqcup_{I_i} \exists_{\vec{x}}([\![Q]\!]_{I_i})$. The result for $P = (\mathbf{abs}\ \vec{x}; c)\, Q$ follows similarly from the continuity of $\forall$ (Lemma 3). $\square$

Example 6 (*Computing the semantics*)
Assume two constraints $\mathtt{out}_a(\cdot)$ and $\mathtt{out}_b(\cdot)$, intuitively representing outputs of names on two different channels a and b. Let $\mathcal{D}$ be the following procedure definitions

$$
\begin{array}{rcl}
\mathcal{D} & = & p() :\!-\ \mathbf{tell}(\mathtt{out}_a(x)) \parallel \mathbf{next}\, \mathbf{tell}(\mathtt{out}_a(y)) \\
& & q() :\!-\ (\mathbf{abs}\ z; \mathtt{out}_a(z))\, (\mathbf{tell}(\mathtt{out}_b(z))) \parallel \mathbf{next}\, q() \\
& & r() :\!-\ p() \parallel q()
\end{array}
$$

The procedure $p()$ outputs on channel a the variables x and y in the first and second time-units respectively. The procedure $q()$ resends on channel b every message received on channel a. The computation of $[\![r()]\!]$ can be found in Figure 3. Let $s \in [\![r()]\!]$. Then, it must be the case that $s \in [\![p()]\!]$ and then, $s(1) \vdash \mathtt{out}_a(x)$ and $s(2) \vdash \mathtt{out}_a(y)$. Since $r \in [\![q()]\!]$, for $i \geq 1$, if $s(i) \vdash \mathtt{out}_a(t)$ then $s(i) \vdash \mathtt{out}_b(t)$ for any term t. Hence, $s(1) \vdash \mathtt{out}_b(x)$ and $s(2) \vdash \mathtt{out}_b(y)$.

3.2 Semantic Correspondence

In this section we prove the soundness and completeness of the semantics.

Lemma 4 (*Soundness*)
Let $[\![\cdot]\!]$ be as in Definition 9. If $P \xrightarrow{(d,d')} R$ and $d \cong d'$, then $d.[\![R]\!] \subseteq [\![P]\!]$.

Proof
Assume that $\langle \vec{x}; P; d\rangle \longrightarrow^* \langle \vec{x}'; P'; d'\rangle \not\longrightarrow$, $\exists \vec{x}(d) \cong \exists \vec{x}'(d')$. We shall prove that $\exists \vec{x}(d).\exists\, \vec{x}'([\![F(P')]\!] \subseteq \exists\, \vec{x}([\![P]\!])$. We proceed by induction on the lexicographical order on the length of the internal derivation and the structure of P, where the predominant component is the length of the derivation. We present the interesting cases. The others can be found in Appendix B.
Case $P = Q \parallel S$. Assume a derivation for $Q = Q_1$ and $S = S_1$ of the form

$$\begin{array}{lll} \langle \vec{z}; Q \parallel S, d\rangle & \longrightarrow^* & \langle \vec{z} \cup \vec{x}_1 \cup \vec{y}_1; Q_1 \parallel S_1, c_1 \sqcup e_1\rangle \\ & \longrightarrow^* & \langle \vec{z} \cup \vec{x}_i \cup \vec{y}_j; Q_i \parallel S_j, c_i \sqcup e_j\rangle \\ & \longrightarrow^* & \langle \vec{z} \cup \vec{x}_m \cup \vec{y}_n; Q_m \parallel S_n; c_m \sqcup e_n\rangle \not\longrightarrow \end{array}$$

such that for $i > 0$, each Q_{i+1} (resp. S_{i+1}) is an evolution of Q_i (resp. S_i); $\vec{x}_i$ (resp. $\vec{y}_j$) are the variables added by Q (resp. S); and c_i (resp e_j) is the information added by Q (resp. S). We assume by alpha-conversion that $\vec{x}_m \cap \vec{y}_n = \emptyset$. We know that $\exists \vec{z}(d) \cong \exists \vec{z}, \vec{x}_m, \vec{y}_n(c_m \sqcup e_n)$ and from $\mathrm{R_{PAR}}$ we can derive:

$$\begin{array}{lll} \langle \vec{z} \cup \vec{y}_n; Q; d \sqcup e_n\rangle & \longrightarrow^* \equiv & \langle \vec{z} \cup \vec{x}_m \cup \vec{y}_n; Q_m, c_m \sqcup e_n\rangle \not\longrightarrow \quad \text{and} \\ \langle \vec{z} \cup \vec{x}_m; S; d \sqcup c_m\rangle & \longrightarrow^* \equiv & \langle \vec{z} \cup \vec{x}_m \cup \vec{y}_n; S_n, c_m \sqcup e_n\rangle \not\longrightarrow \end{array}$$

By (structural) inductive hypothesis, we know that $\exists \vec{z}, \vec{y}_n(d \sqcup e_n).\exists\, \vec{z}, \vec{x}_m, \vec{y}_n[\![F(Q_m)]\!] \subseteq \exists\, \vec{z}, \vec{y}_n([\![Q]\!])$ and also $\exists \vec{z}, \vec{x}_m(d \sqcup c_m).\exists\, \vec{z}, \vec{y}_n, \vec{x}_m[\![F(S_n)]\!] \subseteq \exists\, \vec{z}, \vec{x}_m([\![S]\!])$. We note that $\exists\, \vec{x}([\![P]\!] \cap [\![Q]\!]) = \exists\, \vec{x}([\![P]\!]) \cap [\![Q]\!]$ if $\vec{x} \cap fv(Q) = \emptyset$ (see Proposition 7 in Appendix D). Hence, from the fact that $\vec{x}_m \cap fv(S_n) = \vec{y}_n \cap fv(Q_m) = \emptyset$, we conclude:

$$\exists \vec{z}(d).\exists\, \vec{z}, \vec{x}_m, \vec{y}_n([\![F(Q_m)]\!] \cap [\![F(S_n)]\!]) \subseteq \exists\, \vec{z}([\![Q]\!] \cap [\![S]\!])$$

Case $P = (\mathbf{abs}\ \vec{x}; c)\, Q$. From the rule $\mathrm{R_{ABS}}$, we can show that

$$\begin{array}{ll} \langle \vec{y}; P; d\rangle & \longrightarrow^* \langle \vec{y}_1; P_1 \parallel Q_1^1[\vec{t_1}/\vec{x}]; d_1\rangle \\ & \longrightarrow^* \langle \vec{y}_2; P_2 \parallel Q_1^2[\vec{t_1}/\vec{x}] \parallel Q_2^1[\vec{t_2}/\vec{x}]; d_2\rangle \\ & \longrightarrow^* \langle \vec{y}_3; P_3 \parallel Q_1^3[\vec{t_1}/\vec{x}] \parallel Q_2^2[\vec{t_2}/\vec{x}] \parallel Q_3^1[\vec{t_3}/\vec{x}]; d_3\rangle \\ & \longrightarrow^* \cdots \\ & \longrightarrow^* \langle \vec{y}_n; P_n \parallel Q_1^{m_1}[\vec{t_1}/\vec{x}] \parallel Q_2^{m_2}[\vec{t_2}/\vec{x}] \parallel Q_3^{m_3}[\vec{t_3}/\vec{x}] \parallel \cdots \parallel Q_n^{m_n}[\vec{t_n}/\vec{x}]; d_n\rangle \end{array}$$

where P_n takes the form $(\mathbf{abs}\ \vec{x}; c; E_n)\, Q$, $E_n = \{d_{\vec{x}\vec{t_1}}, ..., d_{\vec{x}\vec{t_n}}\}$ and $\exists \vec{y}(d) \cong \exists \vec{y}_n(d_n)$. Hence, there is a derivation (shorter than that for P) for each $d_{\vec{x}\vec{t_i}} \in E_n$:

$$\langle \vec{y}_i; Q_i^1[\vec{t_i}/\vec{x}]; d_i\rangle \longrightarrow^* \equiv \langle \vec{y}_i'; Q_i^{m_i}[\vec{t_i}/\vec{x}]; d_i'\rangle \not\longrightarrow$$

with $Q[\vec{t_i}/\vec{x}] = Q_i^1[\vec{t_i}/\vec{x}]$ and $\exists \vec{y}_i(d_i) \cong \exists \vec{y}_i'(d_i')$. Therefore, by inductive hypothesis,

$$\exists \vec{y}_i(d_i).\exists\, \vec{y}_i'[\![F(Q_i^{m_i}[\vec{t_i}/\vec{x}])]\!] \subseteq \exists\, \vec{y}_i[\![Q[\vec{t_i}/\vec{x}]]\!]$$

for all $d_{\vec{x}\vec{t_i}} \in E_n$. We assume, by alpha conversion, that the variables added for

each Q_i^j are distinct and then, their intersection is empty. Furthermore, we note that $\exists\vec{y}(d) \cong \exists\vec{y}_1(d_1)$. Since $F(P_n) = \mathbf{skip}$, we then conclude:

$$\exists\vec{y}(d).\exists\,\vec{y}_n[\![F(P_n \parallel \prod_{d_{\vec{x}\vec{t_i}}\in E_n} Q_i^{m_i}[\vec{t_i}/\vec{x}])]\!] \subseteq \exists\,\vec{y}[\![\prod_{d_{\vec{x}\vec{t_i}}\in E_n} Q[\vec{t_i}/\vec{x}])]\!]$$

Let $d.s \in \exists\,\vec{y}[\![\prod_{d_{\vec{x}\vec{t_i}}\in E_n} Q[\vec{t_i}/\vec{x}])]\!]$. For an admissible $d_{\vec{x}\vec{t}}$, either $d \not\vdash c[\vec{t}/\vec{x}]$ or $d \vdash c[\vec{t}/\vec{x}]$.

In the first case, trivially $d.s \in [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]$. In the second case, $E_n \Vdash d_{\vec{x}\vec{t}}$. Hence, $d.s \in [\![Q[\vec{t}/\vec{x}]]\!]$ and $d.s \in [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]$. Here we conclude that for all admissible $[\vec{t}/\vec{x}]$, $d.s \in [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]$ and by Proposition 1 we derive:

$$\exists\vec{y}(d).\exists\,\vec{y}[\![F(\prod_{d_{\vec{x}\vec{t_i}}\in E_n} Q_i^{m_i}[\vec{t_i}/\vec{x}])]\!] \subseteq \exists\,\vec{y}\forall\,\vec{x}[\![(\mathbf{when}\ c\ \mathbf{do}\ Q)]\!]$$

Case $P = p(\vec{t})$. Assume that $p(\vec{x}) : -Q \in \mathcal{D}$. We can verify that

$$\langle\vec{y}; p(\vec{t}); d\rangle \longrightarrow \langle\vec{y}; Q[\vec{t}/\vec{x}]; d\rangle \longrightarrow^* \langle\vec{y}'; Q'; d'\rangle \not\longrightarrow$$

where $\exists\vec{y}'(d') \cong \exists\vec{y}(d)$. By induction $\exists\vec{y}(d).\exists\,\vec{y}'[\![F(Q')]\!] \subseteq \exists\,\vec{y}[\![Q[\vec{t}/\vec{x}]]\!]$ and we conclude $\exists\vec{y}(d).\exists\,\vec{y}[\![F(Q')]\!] \subseteq \exists\,\vec{y}[\![p(\vec{t})]\!]$. □

The previous lemma allows us to prove the soundness of the semantics.

Theorem 3 (*Soundness*)
If $s \in sp(P)$ then there exists s' s.t. $s.s' \in [\![P]\!]$.

Proof
If P is well-terminated under input s, let $s' = \epsilon$. By repeated applications of Lemma 4, $s \in [\![P]\!]$. If P is not well-terminated, then s is finite and let $s' = \mathtt{f}^\omega$ (recall that $\mathtt{f}^\omega$ is quiescent for any process). Via Lemma 4 we can show $s.s' \in [\![P]\!]$. □

Moreover, the semantics approximates any infinite computation.

Corollary 1 (*Infinite Computations*)
Assume that $d.s \in \exists\,\vec{x}_1([\![P_1]\!] \cap \uparrow(c_1.\mathcal{C}^\omega))$ and that $\langle\vec{x}_1; P_1; c_1\rangle \longrightarrow^* \langle\vec{x}_i; P_i; c_i\rangle \longrightarrow^* \langle\vec{x}_n; P_n; c_n\rangle \longrightarrow^* \cdots$. Then, $\bigsqcup \exists\vec{x}_i(c_i) \leq d$.

Proof
Recall that procedure calls must be next guarded. Then, any infinite behavior in P_1 is due to a process of the form $(\mathbf{abs}\ \vec{x}; c)\,Q$ that executes $Q[\vec{t_i}/\vec{x}]$ and adds new information of the form $e[\vec{t_i}/\vec{x}]$. By an analysis similar to that of Lemma 4, we can show that d entails $e[\vec{t_i}/\vec{x}]$. □

Example 7 (*Infinite behavior*)
Let $P = (\mathbf{abs}\ z; \mathtt{out}(z))\,(\mathbf{local}\,x)\,(\mathbf{tell}(\mathtt{out}(x)))$ and let $c = \mathtt{out}(w)$. Starting from the store c, the process P engages in infinitely many internal transitions of the form

$$\langle\emptyset; P; c\rangle \longrightarrow^* \langle\{x_1, \cdots, x_i\}; P_i; \mathtt{out}(x_1) \sqcup \cdots \sqcup \mathtt{out}(x_i) \sqcup \mathtt{out}(w)\rangle \longrightarrow^*$$
$$\langle\{x_1, \cdots, x_i, \cdots, x_n\}; P_n; \mathtt{out}(x_1) \sqcup \cdots \sqcup \mathtt{out}(x_n) \sqcup \mathtt{out}(w)\rangle \longrightarrow^* \cdots$$

At any step of the computation, the observable store is $\mathtt{out}(w) \sqcup \bigsqcup_{i\in 1..n} \exists x_i\,\mathtt{out}(x_i)$ which is equivalent to $\mathtt{out}(w)$. Note also that $\mathtt{out}(w).\mathcal{C}^\omega \in [\![P]\!]$.

For the converse of Theorem 3, we have similar technical problems as in the case of `tcc`, namely: the combination of the **local** operator with the **unless** constructor. Thus, similarly to `tcc`, completeness is verified only for the fragment of `utcc` where there are no occurrences of **unless** processes in the body of **local** processes. The reader may refer (de Boer et al. 1995; Nielsen et al. 2002a) for counterexamples showing that $[\![P]\!] \not\subseteq sp(P)$ when P is not locally independent.

Definition 10 (*Locally Independent Fragment*)
Let $\mathcal{D}.P$ be a program where $\mathcal{D}$ contains process definitions of the form $p_i(\vec{x}) :- P_i$. We say that $\mathcal{D}.P$ is locally independent if for each process of the form $(\mathbf{local}\,\vec{x}; c)\,Q$ in P and P_i it holds that (1) Q does not have occurrences of **unless** processes; and (2) if Q calls to $p_j(\vec{x})$, then P_j satisfies also conditions (1) and (2).

Lemma 5 (*Completeness*)
Let $\mathcal{D}.P$ be a locally independent program s.t. $d.s \in [\![P]\!]$. If $P \xrightarrow{(d,d')} R$ then $d' \cong d$ and $s \in [\![R]\!]$.

Proof
Assume that P is locally independent, $d.s \in [\![P]\!]$ and there is a derivation of the form $\langle \vec{x}; P; d\rangle \longrightarrow^* \langle \vec{x}'; P'; d'\rangle \not\longrightarrow$. We shall prove that $\exists \vec{x}(d) \cong \exists \vec{x}'(d')$ and $s \in \exists\, \vec{x}'[\![F(P')]\!]$. We proceed by induction on the lexicographical order on the length of the internal derivation ($\longrightarrow^*$) and the structure of P, where the predominant component is the length of the derivation. The locally independent condition is used for the case $P = (\mathbf{local}\,\vec{x}; c)\,Q$. We only present the interesting cases. The others can be found in Appendix B.
Case $P = Q \parallel S$. We know that $d.s \in [\![Q]\!]$ and $d.s \in [\![S]\!]$ and by (structural) inductive hypothesis, there are derivations $\langle \vec{z}; Q; d\rangle \longrightarrow^* \langle \vec{z} \cup \vec{x}'; Q'; d' \sqcup c\rangle \not\longrightarrow$ and $\langle \vec{z}; S; d\rangle \longrightarrow^* \langle \vec{z} \cup \vec{y}'; S'; d'' \sqcup e\rangle \not\longrightarrow$ s.t. $s \in \exists\, \vec{z}, \vec{x}'[\![F(Q')]\!]$, $s \in \exists\, \vec{z}, \vec{y}'[\![F(S')]\!]$, $\exists \vec{z}(d) \cong \exists \vec{z}, \vec{x}'(d' \sqcup c)$ and $\exists \vec{z}(d) \cong \exists \vec{z}, \vec{y}'(d'' \sqcup e)$. Therefore, assuming by alpha conversion that $\vec{x}' \cap \vec{y}' = \emptyset$, $\exists \vec{z}(d) \cong \exists \vec{z}, \vec{x}', \vec{y}'(d' \sqcup d'' \sqcup c \sqcup e)$ and by rule $\mathrm{R_{PAR}}$,

$$\langle \vec{z}; Q \parallel S, d\rangle \longrightarrow^* \equiv \langle \vec{z} \cup \vec{x}' \cup \vec{y}'; Q' \parallel S'; d' \sqcup d'' \sqcup c \sqcup e\rangle \not\longrightarrow$$

We note that $\exists\, \vec{x}([\![P]\!] \cap [\![Q]\!]) = \exists\, \vec{x}([\![P]\!]) \cap [\![Q]\!]$ if $\vec{x} \cap fv(Q) = \emptyset$ (see Proposition 7 in Appendix D). Since $F(Q' \parallel S') = F(Q') \parallel F(S')$ and $\vec{x}' \cap fv(S') = \vec{y}' \cap fv(Q') = \emptyset$, we conclude $s \in \exists\, \vec{z}, \vec{x}', \vec{y}'([\![F(Q' \parallel R')]\!])$.
Case $P = (\mathbf{abs}\ \vec{x}; c)\,Q$. By using the rule $\mathrm{R_{ABS}}$ we can show that:

$$\begin{aligned}
\langle \vec{x}; P; d\rangle \quad &\longrightarrow^* \langle \vec{y}_1; P_1 \parallel Q_1^1[\vec{t_1}/\vec{x}]; d_1^1\rangle \\
&\longrightarrow^* \langle \vec{y}_2; P_2 \parallel Q_1^2[\vec{t_1}/\vec{x}] \parallel Q_2^1[\vec{t_2}/\vec{x}]; d_1^2 \sqcup d_2^1\rangle \\
&\longrightarrow^* \langle \vec{y}_3; P_3 \parallel Q_1^3[\vec{t_1}/\vec{x}] \parallel Q_2^2[\vec{t_2}/\vec{x}] \parallel Q_3^1[\vec{t_3}/\vec{x}]; d_1^3 \sqcup d_2^2 \sqcup d_3^1\rangle \\
&\longrightarrow^* \cdots \\
&\longrightarrow^* \langle \vec{y}_n; P_n \parallel Q_1^{m_1}[\vec{t_1}/\vec{x}] \parallel \cdots \parallel Q_n^{m_n}[\vec{t_n}/\vec{x}]; d_1^{m_1} \sqcup \ldots \sqcup d_n^{m_n}\rangle
\end{aligned}$$

where P_n takes the form $(\mathbf{abs}\ \vec{x}; c; E_n)\,Q$ and $E_n = \{d_{\vec{x}\vec{t_1}}, ..., d_{\vec{x}\vec{t_n}}\}$. In the derivation above, d_i^j represents the constraint added by $Q_i^j[\vec{t_i}/\vec{x}]$. Note that $Q[\vec{t_i}/\vec{x}] = Q_i^1[\vec{t_i}/\vec{x}]$. There is a derivation (shorter than that for P) for each $d_{\vec{x}\vec{t_i}} \in E_n$ of the form

$$\langle \vec{y_i}; Q_i^1[\vec{t_i}/\vec{x}]; d_i\rangle \longrightarrow^* \equiv \langle \vec{y_i}'; Q_i^{m_i}[\vec{t_i}/\vec{x}]; d_i^{m_i}\rangle \not\longrightarrow$$

Since $d.s \in [\![P]\!]$, by Proposition 1 we know that $d.s \in [\![Q_i^1[\vec{t_i}/\vec{x}]]\!]$ and by induction, $\exists \vec{y_i}(d_i) \cong \exists \vec{y_i'}(d_i^{m_i})$. Furthermore, it must be the case that $s \in \exists\, \vec{y_i'}[\![F(Q_i^{m_i}[\vec{t_i}/\vec{x}])]\!]$. Let e be the constraint $\exists \vec{y_n}(d_1^{m_1} \sqcup ... \sqcup d_n^{m_n})$. Given that $\exists \vec{y_i}(d_i) \cong \exists \vec{y_i'}(d_i^{m_i})$, we have $\exists \vec{x}(d) \cong e$. Furthermore, given that $F(P_n) = \mathbf{skip}$:

$$(\mathbf{abs}\ \vec{x}; c)\, Q \xrightarrow{(d,e)} (\mathbf{local}\, \vec{y_n})\, F\left(\prod_{d_{\vec{x}\vec{t_i}} \in E_n} Q_i^{m_i}[\vec{t_i}/\vec{x}]\right)$$

Since $s \in \exists\, \vec{y_i'}[\![F(Q_i^{m_i}[\vec{t_i}/\vec{x}])]\!]$ for all $d_{\vec{x}\vec{t_i}} \in E_n$, we conclude

$$s \in \exists\, \vec{y_n}[\![F(\prod_{d_{\vec{x}\vec{t_i}} \in E_n} Q_i^{m_i}[\vec{t_i}/\vec{x}])]\!]$$

Case $P = (\mathbf{local}\, \vec{x})\, Q$. By alpha conversion assume $\vec{x} \notin fv(d.s)$. We know that there exists $d'.s'$ ($\vec{x}$-variant of $d.s$) s.t. $d'.s' \in [\![Q]\!]$, $\exists \vec{x}(d.s) \cong d.s$ and $d.s \cong \exists \vec{x}(d'.s')$. By (structural) inductive hypothesis, there is a derivation $\langle \vec{y}; Q; d' \rangle \longrightarrow^* \langle \vec{y'}; Q'; d'' \rangle \not\longrightarrow$ and $\exists \vec{y}(d') \cong \exists \vec{y'}(d'')$ and $s' \in \exists\, \vec{y'}[\![F(Q')]\!]$. We assume by alpha conversion that $\vec{x} \cap \vec{y} = \emptyset$. Consider now the following derivation:

$$\langle \vec{y}; (\mathbf{local}\, \vec{x})\, Q; d \rangle \longrightarrow \langle \vec{x} \cup \vec{y}; Q; d \rangle \longrightarrow^* \langle \vec{y''}; Q'', c \rangle \not\longrightarrow$$

where $\vec{x} \cup \vec{y} \subseteq \vec{y''}$. We know that $d' \vdash d$ and by monotonicity, we have $\exists \vec{y'}(d'') \vdash \exists \vec{y''}(c)$ and then, $d' \vdash \exists \vec{y''}(c)$. We then conclude $\exists \vec{y}(d) \vdash \exists \vec{y''}(c)$.

Since $s' \in \exists\, \vec{y'}[\![F(Q')]\!]$ then $s \in \exists\, \vec{x} \exists\, \vec{y'}[\![F(Q')]\!]$. Nevertheless, notice that in the above derivation of $(\mathbf{local}\, \vec{x})\, Q$, the final process is Q'' and not Q'. Since Q is monotonic, there are no **unless** processes in it. Furthermore, since $d' \vdash d$, it must be the case that Q' may contain sub-terms (in parallel composition) of the form $R'[\vec{t}/\vec{x}]$ resulting from a process of the form $(\mathbf{abs}\ \vec{y}; e)\, R$ s.t. $d'' \vdash e[\vec{t}/\vec{x}]$ and $c \nvdash e[\vec{t}/\vec{x}]$. Therefore, by Rule $\mathrm{D_{PAR}}$, it must be also the case that $s' \in [\![F(Q'')]\!]$ and then, $s \in \exists\, \vec{x}, \vec{y'}[\![F(Q'')]\!]$. Finally, note that $\vec{y''}$ is not necessarily equal to $\vec{y'}$. With a similar analysis we can show that in Q' there are possibly more **local** processes running in parallel than in Q'' and then, $s \in \exists\, \vec{y''}[\![F(Q'')]\!]$. □

By repeated applications of the previous Lemma, we show the completeness of the denotation with respect to the strongest postcondition relation.

Theorem 4 (*Completeness*)

Let $\mathcal{D}.P$ be a locally independent program, $w = s_1.s_1'$ and $w \in [\![P]\!]$. If $P \xrightarrow{(s_1, s_1')}$ then $s_1 \cong s_1'$. Furthermore, if $P \xrightarrow{(w,w')}_\omega$ then $w \cong w'$.

Notice that completeness of the semantics holds only for the locally independent fragment, while soundness is achieved for the whole language. For the abstract interpretation framework we develop in the next section, we require the semantics to be a sound approximation of the operational semantics and then, the restriction imposed for completeness does not affect the applicability of the framework.

4 Abstract Interpretation Framework

In this section we develop an abstract interpretation framework (Cousot and Cousot 1992) for the analysis of `utcc` (and `tcc`) programs. The framework is based on the above denotational semantics, thus allowing for a compositional analysis. The abstraction proceeds as a composition of two different abstractions: (1) we abstract the constraint system and then (2) we abstract the infinite sequences of *abstract* constraints. The abstraction in (1) allows us to reuse the most popular abstract domains previously defined for logic programming. Adapting those domains, it is possible to perform, e.g., groundness, freeness, type and suspension analyses of `utcc` programs. On the other hand, the abstraction in (2) along with (1) allows for computing the approximated output of the program in a finite number of steps.

4.1 Abstract Constraint Systems

Let us recall some notions from (Falaschi et al. 1997a) and (Zaffanella et al. 1997).

Definition 11 (*Descriptions*)
A description $(\mathcal{C}, \alpha, \mathcal{A})$ between two constraint systems

$$\begin{aligned} \mathbf{C} &= \langle \mathcal{C}, \leq, \sqcup, \mathtt{t}, \mathtt{f}, \mathit{Var}, \exists, D \rangle \\ \mathbf{A} &= \langle \mathcal{A}, \leq^{\alpha}, \sqcup^{\alpha}, \mathtt{t}^{\alpha}, \mathtt{f}^{\alpha}, \mathit{Var}, \exists^{\alpha}, D^{\alpha} \rangle \end{aligned}$$

consists of an abstract domain $(\mathcal{A}, \leq^{\alpha})$ and a surjective and monotonic abstraction function $\alpha : \mathcal{C} \to \mathcal{A}$. We lift α to sequences of constraints in the obvious way.

We shall use c_α, d_α to range over constraints in $\mathcal{A}$ and $s_\alpha, s'_\alpha, w_\alpha, w'_\alpha$, to range over sequences in $\mathcal{A}^*$ and $\mathcal{A}^\omega$ (the set of finite and infinite sequences of constraints in $\mathcal{A}$). To simplify the notation, we omit the subindex "α" when no confusion arises. The entailment $\vdash^{\alpha}$ is defined as in the concrete counterpart, i.e. $c_\alpha \leq^{\alpha} d_\alpha$ iff $d_\alpha \vdash^{\alpha} c_\alpha$. Similarly, $d_\alpha \cong_\alpha c_\alpha$ iff $d_\alpha \vdash^{\alpha} c_\alpha$ and $c_\alpha \vdash^{\alpha} d_\alpha$.

Following standard lines in (Giacobazzi et al. 1995; Falaschi et al. 1997a; Zaffanella et al. 1997) we impose the following restrictions over α relating the cylindrification, diagonal and *lub* operators of $\mathbf{C}$ and $\mathbf{A}$.

Definition 12 (*Correctness*)
Let $\alpha : \mathcal{C} \to \mathcal{A}$ be monotonic and surjective. We say that $\mathbf{A}$ is *upper correct* w.r.t. the constraint system $\mathbf{C}$ if for all $c \in \mathcal{C}$ and $x, y \in Var$:
(1) $\alpha(\exists \vec{x}(c)) \cong_\alpha \exists^{\alpha} \vec{x}(\alpha(c))$.
(2) $\alpha(d_{\vec{xt}}) \cong_\alpha d^{\alpha}_{\vec{xt}}$.
Since α is monotonic, we also have $\alpha(c \sqcup d) \vdash^{\alpha} \alpha(c) \sqcup^{\alpha} \alpha(d)$.

In the example below we illustrate an abstract domain for the groundness analysis of `tcc` programs. Here we give just an intuitive description of it. We shall elaborate more on this domain and its applications in Section 5.2.

Example 8 (*Constraint System for Groundness*)
Let the concrete constraint system $\mathbf{C}$ be the Herbrand constraint system. As abstract constraint system $\mathbf{A}$, let constraints be propositional formulas representing

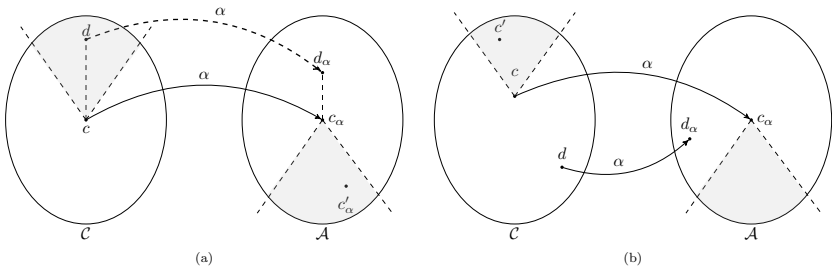

Fig. 4: (a). c'_α approximates c (i.e., $c'_\alpha \propto c$) and $c_\alpha = \alpha(c)$ is the best approximation of c (Definition 13). Since α is monotonic and $c \leq d$, $c_\alpha \leq^\alpha d_\alpha$. In (b), assume that for all d s.t. $d \not\vdash c$, d is not approximated by c_α. Then, all constraint c' approximated by c_α (the upper cone of c) entails c. In this case, $c_\alpha \vdash_{\mathcal{A}} c$ (Definition 14).

$\mathrm{A}_{\mathrm{SKIP}}$	$[\![\mathbf{skip}]\!]^\alpha_X$	$=$	$\mathcal{A}^\omega$
$\mathrm{A}_{\mathrm{TELL}}$	$[\![\mathbf{tell}(c)]\!]^\alpha_X$	$=$	$\uparrow(\alpha(c)).\mathcal{A}^\omega$
$\mathrm{A}_{\mathrm{ASK}}$	$[\![\mathbf{when}\ c\ \mathbf{do}\ P]\!]^\alpha_X$	$=$	$\overline{\Uparrow c}.\mathcal{A}^\omega \cup (\Uparrow c.\mathcal{A}^\omega \cap [\![P]\!]^\alpha_X)$
$\mathrm{A}_{\mathrm{ABS}}$	$[\![(\mathbf{abs}\ \vec{x}; c)\ P]\!]^\alpha_X$	$=$	$\forall\, \vec{x}([\![\mathbf{when}\ c\ \mathbf{do}\ P]\!]^\alpha_X)$
$\mathrm{A}_{\mathrm{PAR}}$	$[\![P \parallel Q]\!]^\alpha_X$	$=$	$[\![P]\!]^\alpha_X \cap [\![Q]\!]^\alpha_X$
$\mathrm{A}_{\mathrm{LOC}}$	$[\![(\mathbf{local}\ \vec{x})\ P]\!]^\alpha_X$	$=$	$\exists\, \vec{x}([\![P]\!]^\alpha_X)$
$\mathrm{A}_{\mathrm{NEXT}}$	$[\![\mathbf{next}\ P]\!]^\alpha_X$	$=$	$\mathcal{A}.[\![P]\!]^\alpha_X$
$\mathrm{A}_{\mathrm{UNL}}$	$[\![\mathbf{unless}\ c\ \mathbf{next}\ P]\!]^\alpha_X$	$=$	$\mathcal{A}^\omega$
$\mathrm{A}_{\mathrm{CALL}}$	$[\![p(\vec{t})]\!]^\alpha_X$	$=$	$X(p(\vec{t}))$

Fig. 5: Abstract denotational semantics for `utcc`. $\vdash_{\mathcal{A}}$ and $\Uparrow$ are in Definition 14. $\overline{A}$ denotes the set complement of A.

groundness information as in $x \wedge (y \leftrightarrow z)$ that means, x is a ground variable and, y is ground iff z is ground. In this setting, $\alpha(x = [a]) = x$ (i.e., x is a ground variable). Furthermore, $\alpha(x = [a|y]) = x \leftrightarrow y$ meaning x is ground if and only if y is ground.

In the following definition we make precise the idea when an abstract constraint approximates a concrete one.

Definition 13 (*Approximations*)
Let $(\mathcal{C}, \alpha, \mathcal{A})$ be a description satisfying the conditions in Definition 11. Given $d_\alpha = \alpha(d)$, we say that d_α is the best approximation of d. Furthermore, for all $c_\alpha \leq^\alpha d_\alpha$ we say that c_α approximates d and we write $c_\alpha \propto d$. This definition is pointwise extended to sequences of constraints in the obvious way (see Figure 4a).

4.2 Abstract Semantics

Now we define an abstract semantics that approximates the observable behavior of a program and is adequate for modular data-flow analysis. The semantic equations are given in Figure 5 and they are parametric on the abstraction function α of the description $(\mathcal{C}, \alpha, \mathcal{A})$. We shall dwell a little upon the description of the rules $\mathrm{A}_{\mathrm{ASK}}$ and $\mathrm{A}_{\mathrm{UNL}}$. The other cases are self-explanatory.

Given the right abstraction of the synchronization mechanism of blocking asks in ccp is crucial to give a safe approximation of the behavior of programs. In abstract interpretation, abstract elements are *weaker* than the concrete ones. Hence, if we approximate the behavior of **when** c **do** P by replacing the guard c with $\alpha(c)$, it could be the case that P proceeds in the abstract semantics but it does not in the concrete one. More precisely, let $d, c \in \mathcal{C}$. Notice that from $\alpha(d) \vdash^{\alpha} \alpha(c)$ we cannot, in general, conclude $d \vdash c$. Take for instance the constraint systems in Example 8. We know that $\alpha(x = a) \cong^{\alpha} \alpha(x = b)$ but $x = a \not\vdash x = b$. Assume now we were to define the abstract semantics of ask processes as:

$$[\![\textbf{when } c \textbf{ do } Q]\!]^{\alpha}_X = \overline{\uparrow(\alpha(c))}.\mathcal{A}^{\omega} \cup (\uparrow(\alpha(c)).\mathcal{A}^{\omega} \cap [\![Q]\!]^{\alpha}_X) \tag{1}$$

A correct analysis of the process $P = \textbf{tell}(x = a) \parallel \textbf{when } x = b \textbf{ do tell}(y = b)$ should conclude that only x is definitely ground. Since $\alpha(x = a) \vdash^{\alpha} \alpha(x = b)$, if we use Equation 1, the analysis ends with the result $(x \wedge y).\mathcal{A}^{\omega}$, i.e., it wrongly concludes that x and y are definitely ground.

We thus follow (Zaffanella et al. 1997; Falaschi et al. 1993; Falaschi et al. 1997a) for the abstract semantics of the ask operator. For this, we need to define the entailment $\vdash_{\mathcal{A}}$ that relates constraints in $\mathcal{A}$ and $\mathcal{C}$.

Definition 14 ($\vdash_{\mathcal{A}}$ *relation*)
Let $d_\alpha \in \mathcal{A}$ and $c \in \mathcal{C}$. We say that d_α entails c, notation $d_\alpha \vdash_{\mathcal{A}} c$, if for all $c' \in \mathcal{C}$ s.t. $d_\alpha \propto c'$ it holds that $c' \vdash c$. We shall use $\Uparrow c$ to denote the set $\{d_\alpha \in \mathcal{A} \mid d_\alpha \vdash_{\mathcal{A}} c\}$.

In words, the (abstract) constraint d_α entails the (concrete) constraint c if all constraints approximated by d_α entail c (see Figure 4b). Then, in Equation $\mathrm{A_{ASK}}$, we guarantee that if the abstract computation proceeds (i.e., $d_\alpha \vdash_{\mathcal{A}} c$) then every concrete computation it approximates proceeds too.

In Equations $\mathrm{D_{ABS}}$ and $\mathrm{D_{LOC}}$ we use the operators $\forall$ and $\exists$ analogous to those in Notation 5. In this context, they are defined on sequences of constraints in $\mathcal{A}^{\omega}$ and they use the elements $\exists^{\alpha}$, $\sqcup^{\alpha}$ and $d^{\alpha}_{\vec{x}\vec{t}}$ instead of their concrete counterparts:

$$\begin{array}{lcl}
\exists\, \vec{x}(S_\alpha) & = & \{s_\alpha \in \mathcal{A}^{\omega} \mid \text{ there exists } s'_\alpha \in S_\alpha \text{ s.t. } \exists^{\alpha}\vec{x}(s_\alpha) \cong_{\alpha} \exists^{\alpha}\vec{x}(s'_\alpha)\} \\
\forall\, \vec{x}(S_\alpha) & = & \{\exists^{\alpha}\vec{y}(s_\alpha) \in S_\alpha \mid \vec{y} \subseteq \mathit{Var}, s_\alpha \in S_\alpha \text{ and for all } s'_\alpha \in \mathcal{A}^{\omega}, \\
 & & \quad \text{if } \exists^{\alpha}\vec{x}(s_\alpha) \cong \exists^{\alpha}\vec{x}(s'_\alpha), (d^{\alpha}_{\vec{x}\vec{t}})^{\omega} \leq s'_\alpha \text{ and } \mathit{adm}(\vec{x}, \vec{t}) \text{ then } s'_\alpha \in S_\alpha\}
\end{array}$$

We omitted the superindex "α" in these operators since it can be easily inferred from the context.

The abstract semantics of the **unless** operator poses similar difficulties as in the case of the ask operator. Moreover, even if we make use of the entailment $\vdash_{\mathcal{A}}$ in Definition 14, we do not obtain a safe approximation. Let us explain this. One could think of defining the semantic equation for the **unless** process as follows:

$$[\![\textbf{unless } c \textbf{ next } Q]\!]^{\alpha}_X = \overline{\Uparrow c}.[\![Q]\!]^{\alpha}_X \cup \Uparrow c.\mathcal{A}^{\omega} \tag{2}$$

The problem here is that $\alpha(d) \not\vdash_{\mathcal{A}} c$ does not imply, in general, $d \not\vdash c$. Take for instance α in Example 8. We know that $x \not\vdash_{\mathcal{A}} x = [a]$ and $x = [a] \vdash x = [a]$. Now let $Q = \textbf{unless } c \textbf{ next tell}(e)$, d be a constraint s.t. $d \vdash c$ and $d_\alpha = \alpha(d)$. We know by rule $\mathrm{D_{UNL}}$ that $d.\mathtt{t}^{\omega} \in [\![Q]\!]$. If $\alpha(d) \not\vdash_{\mathcal{A}} c$, then by using the Equation

(2), we conclude that $d_\alpha.(\mathtt{t}^\alpha)^\omega \notin [\![Q]\!]^\alpha$. Hence, we have a sequence s such that $s \in [\![Q]\!]$ and $\alpha(s) \notin [\![Q]\!]^\alpha$ and the abstract semantics cannot be shown to be a sound approximation of the concrete semantics (see Theorem 5).

Notice that defining $d_\alpha \not\vdash_{\mathcal{A}} c$ as true iff $c' \not\vdash c$ for all c' approximated by d_α does not solve the problem. This is because under this definition, $d_\alpha \not\vdash_{\mathcal{A}} c$ does not hold for any d_α and c. To see this, notice that $\mathtt{f}$ entails all the concrete constraints and it is approximated by any abstract constraint. Therefore, we cannot give a better (safe) approximation of the semantics of **unless** c **next** P than $\mathcal{A}^\omega$ (Rule $\mathrm{A_{UNL}}$).

Now we can formally define the abstract semantics as we did in Section 3. Given a description $(\mathcal{C}, \alpha, \mathcal{A})$, we choose as abstract domain is $\mathbb{A} = (A, \sqsubseteq^\alpha)$ where $A = \{X \mid X \in \mathcal{P}(\mathcal{A}^\omega) \text{ and } (\mathtt{f}^\alpha)^\omega \in X\}$ and $X \sqsubseteq^\alpha Y$ iff $X \supseteq Y$. The bottom and top of this domain are similar to the concrete domain, i.e., $\mathcal{A}^\omega$ and $\{(\mathtt{f}^\alpha)^\omega\}$ respectively.

Definition 15
Let $[\![\cdot]\!]^\alpha_X$ be as in Figure 5. The abstract semantics of a program $\mathcal{D}.P$ is defined as the least fixpoint of the continuous semantic operator:

$$T^\alpha_{\mathcal{D}}(X)(p(\vec{t})) = [\![(Q[\vec{t}/\vec{x}])]\!]^\alpha_X \text{ if } p(\vec{x}) :\!- Q \in \mathcal{D}$$

We shall use $[\![P]\!]^\alpha$ to denote $[\![P]\!]^\alpha_{lfp(T^\alpha_{\mathcal{D}})}$.

The following proposition shows the monotonicity of $[\![\cdot]\!]^\alpha$ and the continuity of $T^\alpha_{\mathcal{D}}$. The proof is analogous to that of Proposition 2.

Proposition 3 (*Monotonicity of $[\![\cdot]\!]^\alpha$ and Continuity of $T^\alpha_{\mathcal{D}}$*)
Let P be a process and $X_1 \sqsubseteq^\alpha X_2 \sqsubseteq^\alpha X_3 ...$ be an ascending chain. Then, $[\![P]\!]^\alpha_{X_i} \sqsubseteq^c [\![P]\!]^\alpha_{X_{i+1}}$ (**Monotonicity**). Moreover, $[\![P]\!]^\alpha_{\bigsqcup_{X_i}} = \bigsqcup_{X_i} [\![P]\!]^\alpha_{X_i}$ (**Continuity**).

4.3 Soundness of the Approximation

This section proves the correctness of the abstract semantics in Definition 15. We first establish a Galois insertion between the concrete and the abstract domains.

Proposition 4 (*Galois Insertion*)
Let $(\mathcal{C}, \alpha', \mathcal{A})$ be a description and $\mathbb{E}$, $\mathbb{A}$ be the concrete and abstract domains. If **A** is upper correct w.r.t. **C** then there exists an upper Galois insertion $\mathbb{E} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{A}$.

Proof
Let $\mathbb{A} = (A, \sqsubseteq^\alpha)$, $\mathbb{E} = (E, \sqsubseteq^c)$ and $\alpha : E \to A$ and $\gamma : A \to E$ be defined as follows:

$$\begin{aligned} \alpha(S) &= \{\beta(s) \mid s \in S\} \text{ for } S \in \{X \mid X \in \mathcal{P}(\mathcal{C}^\omega) \text{ and } \mathtt{f}^\omega \in X\} \\ \gamma(S_\alpha) &= \{s \mid \beta(s) \in S_\alpha\} \text{ for } S_\alpha \in \{X \mid X \in \mathcal{P}(\mathcal{A}^\omega) \text{ and } (\mathtt{f}^\alpha)^\omega \in X\} \end{aligned}$$

where β is the pointwise extension of α' over sequences. Notice that β is a monotonic and surjective function between $\mathcal{C}^\omega$ and $\mathcal{A}^\omega$ and set intersection is the lub in both $\mathbb{E}$ and $\mathbb{A}$. We conclude by the fact that any additive and surjective function between complete lattices defines a Galois insertion (Cousot and Cousot 1979). □

We lift, as standardly done in abstract interpretations (Cousot and Cousot 1992), the approximation induced by the above abstraction. Let $I : ProcHeads \to E$, $X : ProcHeads \to A$, β be as in Proposition 4 and p be a process definition. Then

$$\alpha(I(p)) = \{\beta(s) \mid s \in I(p)\} \qquad \gamma(X(p)) = \{s \mid \beta(s) \in X(p)\}$$

We conclude here by showing that concrete computations are safely approximated by the abstract semantics.

Theorem 5 (*Soundness of the approximation*)
Let $(\mathcal{C}, \alpha, \mathcal{A})$ be a description and **A** be upper correct w.r.t. **C**. Given a `utcc` program $\mathcal{D}.P$, if $s \in [\![P]\!]$ then $\alpha(s) \in [\![P]\!]^\alpha$.

Proof
Let $d_\alpha.s_\alpha = \alpha(d.s)$ and assume that $d.s \in [\![P]\!]$. Then, $d.s \in [\![P]\!]_I$ where I is the lfp of $T_\mathcal{D}$. By the continuity of $T_\mathcal{D}$, there exists n s.t. $I = T^n_\mathcal{D}(I_\perp)$ (the n-th application of $T_\mathcal{D}$). We proceed by induction on the lexicographical order on the pair n and the structure of P, where the predominant component is n. We only present the interesting cases. The others can be found in Appendix C.
Case $P = (\mathbf{abs}\ \vec{x}; c)\, Q$. Let $[\vec{t}/\vec{x}]$ be an admissible substitution. We shall prove that $s \in [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]$ implies $s_\alpha \in [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]^\alpha$. The result follows from Proposition 1 and from the fact that $s_\alpha \in \forall\, \vec{x}([\![\mathbf{when}\ c\ \mathbf{do}\ Q]\!]^\alpha)$ iff $s_\alpha \in [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]^\alpha$ for all $adm(\vec{x}, \vec{t})$. The proof of the previous statement is similar to that of Proposition 1 and it appears in Appendix D.

Assume that $d \vdash c[\vec{t}/\vec{x}]$. Then, $d.s \in [\![Q[\vec{t}/\vec{x}]]\!]$ and we distinguish two cases:
(1) $d_\alpha \vdash_\mathcal{A} c[\vec{t}/\vec{x}]$. Since $d.s \in [\![Q[\vec{t}/\vec{x}]]\!]$ then $d.s \in \exists\, \vec{x}([\![Q]\!] \cap \uparrow(d^\omega_{\vec{x}\vec{t}}))$. Therefore, there exists $d'.s'$, an $\vec{x}$-variant of $d.s$, s.t. $d'.s' \in [\![Q]\!]$ and $d'.s' \in \uparrow(d^\omega_{\vec{x}\vec{t}})$. By (structural) inductive hypothesis, $\alpha(d'.s') \in [\![Q]\!]^\alpha$. Furthermore, by monotonicity of α and Property (2) in Definition 12, we derive $\alpha(d'.s') \in \uparrow (d^\alpha_{\vec{x}\vec{t}})^\omega$. Hence $\alpha(d'.s') \in ([\![Q]\!]^\alpha \cap \uparrow((d^\alpha_{\vec{x}\vec{t}})^\omega)$. Since $\exists \vec{x}(d.s) = \exists \vec{x}(d'.s')$, by Property (1) in Definition 12, we have $\exists^\alpha \vec{x}(\alpha(d.s)) = \exists^\alpha \vec{x}(\alpha(d'.s'))$ (i.e., $\alpha(d'.s')$ is an $\vec{x}$-variant of $d_\alpha.s_\alpha$). Then, $d_\alpha.s_\alpha \in \exists\, \vec{x}([\![Q]\!]^\alpha \cap \uparrow((d^\alpha_{\vec{x}\vec{t}})^\omega))$ and we conclude $d_\alpha.s_\alpha \in [\![Q[\vec{t}/\vec{x}]]\!]^\alpha$.
(2) $d_\alpha \not\vdash_\mathcal{A} c[\vec{t}/\vec{x}]$. Hence trivially $d_\alpha.s_\alpha \in [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]^\alpha$.

We conclude by noticing that if $d \not\vdash c[\vec{t}/\vec{x}]$ then $d_\alpha \not\vdash_\mathcal{A} c[\vec{t}/\vec{x}]$ and therefore $d_\alpha.s_\alpha \in [\![(\mathbf{when}\ c\ \mathbf{do}\ Q)[\vec{t}/\vec{x}]]\!]^\alpha$.
Case $P :\!- p(\vec{t})$. Let $p(\vec{x}) :\!- Q$ in $\mathcal{D}$ be a process definition. If $d.s \in [\![p(\vec{t})]\!]$ then $d.s \in I(p(\vec{t}))$ (recall that $I = lfp(T_\mathcal{D})$). We know that $d.s \in [\![Q[\vec{t}/\vec{x}]]\!]$ and then, $d.s \in [\![Q[\vec{t}/\vec{x}]]\!]_{I'}$ where $I' = T^m_\mathcal{D}(I_\perp)$ with $m < n$. By induction, and continuity of $T^\alpha_\mathcal{D}$, we know that $d_\alpha.s_\alpha \in [\![Q[\vec{t}/\vec{x}]]\!]^\alpha$ and then $d_\alpha.s_\alpha \in [\![p(\vec{t})]\!]^\alpha$. □

4.4 Obtaining a finite analysis

As standard in Abstract Interpretation, it is possible to obtain an analysis which terminates, by imposing several alternative conditions (see for instance Chapter 9 in (Cousot and Cousot 1992)). So, one possibility is to impose that the abstract domain is noetherian (also called finite ascending chain condition). Another possibility is to

use widening operators, or to find an abstract domain that guarantees termination after a finite number of steps. So, our framework allows to use all this classical methodologies. In the examples that we have developed we shall focus our attention on a special class of abstract interpretations obtained by defining what we call a *sequence abstraction* mapping possibly infinite sequences of (abstract) constraints into finite ones. Actually we can define these abstractions as Galois connections.

Definition 16 (*k-sequence Abstraction*)
A k-sequence abstraction is given by the following pair of functions (α_k, γ_k), with $\alpha_k : (\mathcal{A}^\omega, \leq^\alpha) \to (\mathcal{A}_k^*, \leq^\alpha)$, and $\gamma_k : (\mathcal{A}_k^*, \leq^\alpha) \to (\mathcal{A}^\omega, \leq^\alpha)$. As for the function α_k, we set $\alpha_k(s) = s'$ where s' has length k and $s'(i) = s(i)$ for $i \leq k$. Similarly, $\gamma_k(s') = s$ where $s'(i) = s(i)$ for $i \leq k$ and $s'(i) = \mathtt{t}$ for $i > k$.

It is easy to see that, for any k, (α_k, γ_k) defines a Galois connection between $(\mathcal{A}^\omega, \leq^\alpha)$ and $(\mathcal{A}_k^*, \leq^\alpha)$. Thus it is possible to use compositions of Galois connections for obtaining a new abstraction (Cousot and Cousot 1992).

If $\mathcal{A}$ in $(\mathcal{C}, \alpha, \mathcal{A})$ leads to a Noetherian abstract domain $\mathbb{A}$, then the abstraction obtained from the composition of α and any α_k above guarantees that the fixpoint of the abstract semantics can be reached in a finite number of iterations. Actually the domain that we obtain in this way is given by sequences cut at length k. The number k determines the length of the cut and hence the precision of the approximation. The bigger k the better the approximation.

5 Applications

This section is devoted to show some applications of the abstract semantics developed here. We shall describe three specific abstract domains as instances of our framework: (1) we abstract a constraint system representing cryptographic primitives. Then we use the abstract semantics to exhibit a secrecy flaw in a security protocol modeled in `utcc`. Next, (2) we tailor two abstract domains from logic programming to perform a groundness and a type analysis of a `tcc` program. We then apply this analysis in the verification of a reactive system in `tcc`. Finally, (3) we propose an abstract constraint system for the suspension analysis of `tcc` programs.

5.1 Verification of Security Protocols

The ability of `utcc` to express mobile behavior, as in Example 2, allows for the modeling of security protocols. Here we describe an abstraction of a cryptographic constraint system in order to bound the length of the messages to be considered in a secrecy analysis. We start by recalling the constraint system in (Olarte and Valencia 2008b) whose terms represent the messages generated by the protocol and cryptographic primitives are represented as functions over such terms.

Definition 17 (*Cryptographic Constraint System*)
Let Σ be a signature with constant symbols in $\mathcal{P} \cup \mathcal{K}$, function symbols *enc*, *pair*, *priv* and *pub* and predicates $\mathtt{out}(\cdot)$ and $\mathtt{secret}(\cdot)$. Constraint in $\mathcal{C}$ are formulas built from predicates in Σ, conjunction ($\sqcup$) and $\exists$.

Intuitively, $\mathcal{P}$ and $\mathcal{K}$ represent respectively the principal identifiers, e.g. $A, B, \ldots$ and keys k, k'. We use $\{m\}_k$ and (m_1, m_2) respectively, for $enc(m, k)$ (encryption) and $pair(m_1, m_2)$ (composition). For the generation of keys, $priv(k)$ stands for the private key associated to the value k and $pub(k)$ for its public key.

As standardly done in the verification of security protocols, a Dolev-Yao attacker (Dolev and Yao 1983) is presupposed, able to eavesdrop, disassemble, compose, encrypt and decrypt messages with available keys. The ability to eavesdrop all the messages in transit in the network is implicit in our model due to the shared store of constraints. The other abilities are modeled by the following `utcc` processes:

$$
\begin{array}{lll}
Disam() & :- & (\mathbf{abs}\ x, y; \mathtt{out}(\ (x, y)\ ))\, \mathbf{tell}(\mathtt{out}\,(x) \sqcup \mathtt{out}\,(y)) \\
Comp() & :- & (\mathbf{abs}\ x, y; \mathtt{out}(x) \sqcup \mathtt{out}(y))\, \mathbf{tell}(\mathtt{out}\,(\ (x, y)\ )) \\
Enc() & :- & (\mathbf{abs}\ x, y; \mathtt{out}(x) \sqcup \mathtt{out}(y))\, \mathbf{tell}(\mathtt{out}\,(\{x\}_{pub(y)})) \\
Dec() & :- & (\mathbf{abs}\ x, y; \mathtt{out}(priv(y)) \sqcup \mathtt{out}(\{x\}_{pub(y)}))\, \mathbf{tell}(\mathtt{out}\,(x)) \\
Pers() & :- & (\mathbf{abs}\ x; \mathtt{out}(x))\, \mathbf{next}\, \mathbf{tell}(\mathtt{out}(x)) \\
Spy() & :- & Disam() \parallel Comp() \parallel Enc() \parallel Dec() \parallel Pers() \parallel \mathbf{next}\, Spy()
\end{array}
$$

Since the final store is not automatically transferred to the next time-unit, the process $Pers$ above models the ability to remember all messages posted so far.

It is easy to see that the process $Spy()$ in a store $\mathtt{out}(m)$ may add messages of unbounded length. Take for example the process $Comp()$ that will add the constraints $\mathtt{out}(m)$, $\mathtt{out}((m, (m, m)))$, $\mathtt{out}(((m, m), m))$ and so on.

To deal with the inherent state explosion problem in the model of the attacker, symbolic (compact) representations of the behavior of the attacker have been proposed, for instance in (Boreale 2001; Fiore and Abadi 2001; Olarte and Valencia 2008b; Bodei et al. 2010). Here we follow the approach of restricting the number of states to be considered in the verification of the protocol, as for instance in (Escobar et al. 2011; Song et al. 2001; Armando and Compagna 2008). Roughly, we shall cut the messages generated of length greater than a given κ, thus allowing us to model a bounded version of the attacker.

Before defining the abstraction, we notice that the constraint system we are considering includes existentially quantified syntactic equations. For this kind of equations it is necessary to refer to a solved form of them in order to have a uniform way to compute an approximation of the constraint system. We then consider constraints of the shape $\exists \vec{y}(x_1 = t_1(\vec{y}) \sqcup \ldots \sqcup x_n = t_n(\vec{y}))$ where $\vec{x} = x_1, \ldots x_n$ are pairwise distinct and $\vec{x} \cap \vec{y} = \emptyset$. Here, $t(\vec{y})$ refers to a term where $fv(t(\vec{y})) \subseteq \vec{y}$. Given a constraint, its normal form can be obtained by applying the algorithm proposed in (Maher 1988) where: quantifiers are moved to the outermost position and equations of the form $f(t_1, \ldots, t_n) = f(t'_1, \ldots, t'_n)$ are replaced by $t_1 = t'_1 \sqcup \ldots \sqcup t_n = t'_n$; equations such as $x = x$ are deleted; equation of the form $t = x$ are replaced by $x = t$; and given $x = t$, if x does not occur in t, x is replaced by t in t' in all equation of the form $x' = t'$. For instance, the solved form of $\exists z, y(x = f(y) \sqcup y = g(z))$ is the constraint $\exists z(x = f(g(z)))$.

Definition 18 (*Abstract secure constraint system*)
Let $\mathcal{M}$ be the set of terms (messages) generated from the signature Σ in Definition 17. Let $lg : \mathcal{M} \to \mathbb{N}$ be defined as $lg(m) = 0$ if $m \in \mathcal{P} \cup \mathcal{K} \cup Var$; $lg(\{m_1\}_{m_2}) =$

$$\begin{array}{lcll} \mathsf{M}_1 & A \to B & : & \{(m, A)\}_{pub(B)} \\ \mathsf{M}_2 & B \to A & : & \{(m, n, B)\}_{pub(A)} \\ \mathsf{M}_3 & A \to B & : & \{n\}_{pub(B)} \end{array}$$

(a)

$$\begin{array}{lcll} \mathsf{M}_1 & A \to C & : & \{(m, A)\}_{pub\,(C)} \\ \mathsf{M}'_1 & C \to B & : & \{(m, A)\}_{pub\,(B)} \\ \mathsf{M}_2 & B \to A & : & \{(m, n, B)\}_{pub\,(A)} \\ \mathsf{M}_3 & A \to C & : & \{n\}_{pub\,(C)} \end{array}$$

(b)

Fig. 6: Steps of the Needham-Schroeder Protocol

$lg(\ (m_1, m_2)\ ) = 1 + lg(m_1) + lg(m_2)$. Let $cut_\kappa(m) = m$ if $lg(m) \leq \kappa$. Otherwise, $cut_\kappa(m) = m_\top$ where $m_\top \notin \mathcal{M}$ represents all the messages whose length is greater than κ. We define $\alpha(c)$ as $\alpha_\kappa(NF(c))$ where

$$\begin{array}{lcllcl} \alpha_\kappa(c(m)) & = & c(cut_\kappa(m)) & \alpha_\kappa(d_{xt}) & = & d_{xt'} \text{ where } t' = cut_\kappa(t) \\ \alpha_\kappa(c \sqcup c') & = & \alpha_\kappa(c) \sqcup \alpha_\kappa(c') & \alpha_\kappa(\exists \vec{x} c) & = & \exists \vec{x} \alpha_\kappa(c) \end{array}$$

and $NF(c)$ is a solved form of the constraint c. We omit the superscript α in the abstract operators $\sqcup^\alpha$, $\exists^\alpha$ and $d^\alpha_{\vec{x}\vec{t}}$ to simplify the notation.

We note that the previous abstraction reminds of the *depth-κ* abstractions typically done in the analysis of logic programs (see e.g., (Sato and Tamaki 1984)).

We shall illustrate the use of the abstract constraint system above by performing a secrecy analysis on the Needham-Schröder (NS) protocol (Lowe 1996). This protocol aims at distributing two *nonces* in a secure way. Figure 6(a) shows the steps of NS where m and n represent the nonces generated, respectively, by the principals A and B. The protocol initiates when A sends to B a new nonce m together with her own agent name A, both encrypted with B's public key. When B receives the message, he decrypts it with his secret private key. Once decrypted, B prepares an encrypted message for A that contains a new nonce n together with the nonce m and his name B. A then recovers the clear text using her private key. A convinces herself that this message really comes from B by checking whether she got back the same nonce sent out in the first message. If that is the case, she acknowledges B by returning his nonce. B does a similar test.

Assume the execution of the protocol in Figure 6(b). Here C is an intruder, i.e. a malicious agent playing the role of a principal in the protocol. As it was shown in (Lowe 1996), this execution leads to a secrecy flaw where the attacker C can reveal n which is meant to be known only by A and B. In this execution, the attacker replies to B the message sent by A and B believes that he is establishing a session key with A. Since the attacker knows the private key $priv(C)$, she can decrypt the message $\{n\}_{pub\,(C)}$ and n is no longer a secret between B and A as intended.

We model the behavior of the principals of the NS protocol with the process definitions in Figure 7. Nonce generation is modeled by **local** constructs and the process **tell**(`out`(m)) models the broadcast of the message m. Inputs (message reception) are modeled by **abs** processes as in Example 3. In *Resp*, we use the process $Secrete(n)$ to state that the nonce n cannot be revealed. Finally, the process $SpKn$ corresponds to the initial knowledge of the attacker: the names of the principals, their public keys and the leaked keys in the set Bad (e.g., the private key of C in the configuration of Figure 6 (b)).

$$
\begin{array}{lcl}
Init(i,r) & :- & (\mathbf{local}\, m)\, \mathbf{tell}(\mathtt{out}(\{(m,i)\}_{pub(r))}) \parallel \\
 & & \qquad \mathbf{next}\, (\mathbf{abs}\, x; \mathtt{out}(\{(m,x,r)\}_{pub(i)}))\, \mathbf{tell}(\mathtt{out}(\{x\}_{pub(r))}) \\
 & & \parallel \mathbf{next}\, Init(i,r) \\
Resp(r) & :- & (\mathbf{abs}\, x,u; \mathtt{out}(\{(x,u)\}_{pub(r)}))\, \mathbf{next} \\
 & & \qquad (\mathbf{local}\, n)\, (Secrete(n) \parallel \mathbf{tell}(\mathtt{out}(\{x,n,r\}_{pub\,(u))})) \\
 & & \parallel \mathbf{next}\, Resp(r) \\
Secrete(x) & :- & \mathbf{tell}(\mathtt{secret}(x)) \parallel \mathbf{next}\, Secrete(x) \\
SpKn() & :- & \parallel_{A \in \mathcal{P}}\ \mathbf{tell}(\mathtt{out}(A) \sqcup \mathtt{out}(pub(A))) \\
 & & \parallel_{A \in Bad}\ \mathbf{tell}(\mathtt{out}(priv(A))) \\
 & & \parallel \mathbf{next}\, SpKn()
\end{array}
$$

Fig. 7: `utcc` model of the Needham-Schröder Protocol

$$
\begin{array}{lcl}
[\![Init(A,C)]\!]^\alpha & = & \exists\ m_1\ \exists\ m_2\ \ (\{c_1.c_2 \mid c_1 \vdash^\alpha \mathtt{out}(\{m_1, A\}_{pub\,(C)}),\ c_2 \vdash^\alpha \mathtt{out}(\{A, m_2\}_{pub\,(C)})\} \cap \\
 & & \quad \mathcal{A}.\forall\, x(\{c_2 \mid \text{if } c_2 \vdash_{\mathcal{A}} \mathtt{out}(\{m_1, x, C\}_{pub(A)}) \text{ then } c_2 \vdash^\alpha \mathtt{out}(\{x\}_{pub(C)})\})) \\
[\![resp(B)]\!]^\alpha & = & \forall\, x,u(\exists\ n_1\ \{c_1.c_2 \mid \text{if } c_1 \vdash_{\mathcal{A}} \mathtt{out}(\{x,u\}_{pub(B)}) \text{ then} \\
 & & \quad c_2 \vdash^\alpha \mathtt{secret}(n_1) \sqcup \mathtt{out}(\{x, n_1, B\}_{pub(u)})\}) \\
[\![Spy]\!]^\alpha & = & \forall\ x\ (\{c_1.c_2 \mid \text{ if } c_1 \vdash_{\mathcal{A}} \mathtt{out}(x) \text{ then } c_2 \vdash^\alpha \mathtt{out}(x)\}) \cap S.S \text{ where} \\
S & = & \forall\, x,y(\{c \mid \text{ if } c \vdash_{\mathcal{A}} \mathtt{out}(x) \sqcup \mathtt{out}(y) \text{ then } c \vdash^\alpha \mathtt{out}(\{x,y\}) \sqcup \mathtt{out}(\{x\}_{pub(y)})\} \cap \\
 & & \quad \{c \mid \text{ if } c \vdash_{\mathcal{A}} \mathtt{out}(\{x,y\}) \text{ then } c \vdash^\alpha \mathtt{out}(x) \sqcup \mathtt{out}(y)\} \cap \\
 & & \quad \{c \mid \text{ if } c \vdash_{\mathcal{A}} \mathtt{out}(\{x\}_{pub(y)}) \sqcup \mathtt{out}(priv(y)) \text{ then } c \vdash^\alpha \mathtt{out}(x)\}) \\
[\![SpKn]\!]^\alpha & = & \{c_1.c_2 \mid c_i \vdash^\alpha \mathtt{out}(pub(A)) \sqcup \mathtt{out}(pub(B)) \sqcup \mathtt{out}(pub(C))\} \cap \\
 & & \{c_1.c_2 \mid c_i \vdash^\alpha \mathtt{out}(A) \sqcup \mathtt{out}(B) \sqcup \mathtt{out}(C) \sqcup \mathtt{out}(priv(C))\} \\
[\![NS]\!]^\alpha & = & [\![Spy]\!]^\alpha \cap [\![SpKn]\!]^\alpha \cap [\![\mathbf{init}(A,C)]\!]^\alpha \cap [\![\mathbf{resp}(B)]\!]^\alpha
\end{array}
$$

Fig. 8: Abstract semantics of the process NS in Equation 3

Consider the following process:

$$NS :- \ \ Spy \parallel SpKn \parallel Init(A,C) \parallel Resp(B) \tag{3}$$

By using the composition of α_3 (as in Definition 18) and the sequence abstraction 2-*sequence*, we obtain the abstract semantics of NS as showed in Figure 8. This allows us to exhibit the secrecy flaw of the NS protocol pointed out in (Lowe 1996): Let $s = c_1.c_2$ s.t. $s \in [\![\mathbf{NS}]\!]^\alpha$. Then, there exist a m_1-n_1-variant $s' = c_1'.c_2'$ of s s.t.

$$
\begin{array}{l}
c_1' \vdash \mathtt{out}(\{m_1, A\}_{pub(C)}) \sqcup \mathtt{out}(priv(C)) \sqcup \mathtt{out}(\{m_1, A\}_{pub(B)}) \\
c_2' \vdash \mathtt{out}(\{m_1, n_1, A\}_{pub(A)}) \sqcup \mathtt{out}(\{n_1\}_{pub(C)}) \sqcup \mathtt{out}(\mathtt{secret}(n_1)) \sqcup \mathtt{out}(\mathtt{out}(n_1))
\end{array}
$$

This means that the nonce n_1 appears as plain text in the network and it is no longer a secret between A and B as intended.

5.2 Groundness Analysis

In logic programming one useful analysis is groundness. It aims at determining if a variable will always be bound to a ground term. This information can be used, e.g., for optimization in the compiler or as base for other data flow analyses such as independence analysis, suspension analysis, etc. Here we present a groundness anal-

$gen_a(x) \quad :- \quad (\mathbf{local}\, x')\,(assign(x,[a|x']) \parallel$
$\qquad\qquad \mathbf{when}\ go_a = [] \ \mathbf{do\ next}\ gen_a(x') \parallel \mathbf{when}\ stop_a = [] \ \mathbf{do}\ assign(x',[]))$

$assign(x,y) \quad :- \quad \mathbf{tell}(x=y) \parallel \mathbf{next}\ assign(x,y)$

$append(x,y,z) \quad :- \quad \mathbf{when}\ x = [] \ \mathbf{do}\ assign(y,z) \parallel$
$\mathbf{when}\ \exists_{x',x''}(x = [x'\,|x'']) \ \mathbf{do}$
$(\mathbf{local}\, x', x'', z')\,(assign(x,[x'|x'']) \parallel assign(z,[x'|z']) \parallel \mathbf{next}\ append(x'',y,z'))$

Fig. 9: Appending streams (Example 9). The process definition gen_b is similar to gen_a but replacing the constant a with b.

ysis for a `tcc` program. To this end, we shall use as concrete domain the Herbrand Constraint System and the following running example.

Example 9 (*Append*)
Assume the process definitions in Figure 9. The process $gen_a(x)$ adds an "a" to the stream x when the environment provides $go_a = []$ as input. Under input $stop_a = []$, $gen_a(x)$ terminates the stream binding its tail to the empty list. The process gen_b can be explained similarly. The process $assign(x,y)$ persistently equates x and y. Finally, $append(x,y,z)$ binds z to the concatenation of x and y.

We shall use Pos (Armstrong et al. 1998) as abstract domain for the groundness analysis. In Pos, positive propositional formulas represent groundness dependencies among variables. For instance, $\alpha_G(x = [a|b]) = x$ meaning that x is a ground variable and $\alpha_G(x = [y|z]) = x \leftrightarrow (y \wedge z)$ meaning that x is ground if and only if both y and z are ground. Elements in this domain are ordered by logical implication, e.g., $x \sqcup (x \leftrightarrow (y \wedge z)) \vdash_{\alpha_G} y$.

Observation 2 (*Precision of Pos with respect to Synchronization*)
Notice that Pos does not distinguish between the empty list and a list of ground terms: $d_\kappa = \alpha_G(x = []) = \alpha_G(x = [a]) = x$ and then, $d_\kappa \not\vdash_{\mathcal{A}} x = []$ (see Definition 14). This affects the precision of the analysis. For instance, let $P = \mathbf{tell}(x = [])$ and $Q = \mathbf{when}\ x = []\ \mathbf{do\ tell}(y = [])$. One would expect that the groundness analysis of $P \parallel Q$ determines that x and y are ground variables. Nevertheless, it is easy to see that $x.true^\omega \in [\![P]\!]^{\alpha_G}$ and then, the information added by $\mathbf{tell}(y = [])$ is lost.

We improve the accuracy of the analysis by using the abstract domain defined in (Codish and Demoen 1994) to derive information about type dependencies on terms. The abstraction is defined as follows:

$$\alpha_T(x = t) = \begin{cases} list(x, x_s) & \text{if} \quad t = [y \mid x_s] \text{ for some } y \\ nil(x) & \text{if} \quad t = [] \end{cases}$$

Informally, $list(x, x_s)$ means x is a list iff x_s is a list and $nil(x)$ means x is the empty list. If x is a list we write $list(x)$ and $nil(x) \vdash^{\alpha_T} list(x)$. Elements in the domain are ordered by logical implication.

The following constraint systems result from the reduced product (Cousot and Cousot 1992) of the previous abstract domains, thus allowing us to capture groundness and type dependency information.

$[\![gen_a(x) \parallel gen_b(y) \parallel append(x,y,z)]\!]^\alpha = \exists\, x_1(GA_1) \cap \exists\, y_1(GB_1) \cap A_1$ where

$$
\begin{array}{lcl}
GA_1 & = & \uparrow\langle x \leftrightarrow x_1, list(x,x_1)\rangle.\mathcal{A} \cap \\
 & & \{c.s \mid \text{if } c \vdash_{\mathcal{A}} go_a = [] \text{ then } s \in \exists\, x_2(GA_2)\} \cap \\
 & & \{c.s \mid \text{if } c \vdash_{\mathcal{A}} stop_a = [] \text{ then } \langle x_1, nil(x_1)\rangle^\omega \leq^\alpha c.s\} \\
\ldots & & \\
GA_\kappa & = & \uparrow\langle x_{\kappa-1} \leftrightarrow x_\kappa, list(x_{\kappa-1}, x_\kappa)\rangle.\epsilon \cap \\
 & & \{c.\epsilon \mid \text{if } c \vdash_{\mathcal{A}} stop_a = [] \text{ then } c \vdash^\alpha \langle x_\kappa, nil(x_\kappa)\rangle\} \\
 & & \\
A_1 & = & \{c.s \mid \text{if } c \vdash_{\mathcal{A}} x = [] \text{ then } (d^\alpha_{yz})^\omega \leq^\alpha c.s\} \cap \\
 & & \{c.s \mid \text{if } c \vdash_{\mathcal{A}} \exists x', x_2(x = [x'|x_2]) \text{ then} \\
 & & \qquad c.s \in \exists\, x' \exists\, x_2 \exists\, z_2(\uparrow(\langle x \leftrightarrow x_2, list(x,x_2)\rangle^\omega) \cap \\
 & & \qquad\qquad \uparrow(\langle z \leftrightarrow z_2, list(z,z_2)\rangle^\omega) \cap \mathcal{A}.A_2)\} \\
\ldots & & \\
A_\kappa & = & \{c.\epsilon \mid \text{if } c \vdash_{\mathcal{A}} x_\kappa = [] \text{ then } d^\alpha_{y_\kappa z_\kappa} \leq^\alpha c\} \cap \\
 & & \{c.\epsilon \mid \text{if } c \vdash_{\mathcal{A}} \exists x', x_{\kappa'}(x = [x'|x_{\kappa'}]) \text{ then} \\
 & & \qquad c.\epsilon \in \exists\, x' \exists\, x_{\kappa'} \exists\, z_{\kappa'}(\uparrow(\langle x_\kappa \leftrightarrow x_{\kappa'}, list(x_\kappa, x_{\kappa'})\rangle).\epsilon \cap \\
 & & \qquad\qquad \uparrow(\langle z_\kappa \leftrightarrow z_{\kappa'}, list(z_\kappa, z_{\kappa'})\rangle).\epsilon)\}
\end{array}
$$

Fig. 10: Abstract semantics of the process $P = gen_a(x) \parallel gen_b(y) \parallel append(x,y,z)$. Definitions of $gen_a(x), gen_b(y)$ and $append(x,y,z)$ are given in Example 9. Sets $GB_1, .., GB_\kappa$ are similar to $GA_1, .., GA_\kappa$ and omitted here.

Definition 19 (*Groundness-type Constraint System*)
Let $\mathbf{A_{GT}} = \langle \mathcal{A}, \leq^{\alpha_{GT}}, \sqcup^{\alpha_{GT}}, \mathtt{t}^{\alpha_{GT}}, \mathtt{f}^{\alpha_{GT}}, Var, \exists^{\alpha_{GT}}, d^{\alpha_{GT}}\rangle$. Given $c \in \mathcal{C}$, $\alpha_{GT}(c) = \langle \alpha_G(c), \alpha_T(c)\rangle$. The operations $\sqcup^{\alpha_{GT}}$ and $\exists^{\alpha_{GT}}$ correspond to logical conjunction and existential quantification on the components of the tuple and $d^{\alpha_{GT}}_{\vec{x}\vec{t}}$ is defined as $\langle \alpha_G(\vec{x} = \vec{t}), \alpha_T(\vec{x} = \vec{t})\rangle$. Finally, $\langle c_\kappa, d_\kappa\rangle \leq^{\alpha_{GT}} \langle c'_\kappa, d'_\kappa\rangle$ iff $c'_\kappa \vdash_{\alpha_G} c_\kappa$ and $d'_\kappa \vdash_{\alpha_T} d_\kappa$.

Consider the Example 9 and the abstraction α resulting from the composition of α_{GT} above and $sequence_\kappa$. Note that the program makes use of guards of the form $\exists x', x''(x = [x'|x''])$ and $x = []$. Note also that $list(x,x') \vdash_{\mathcal{A}} \exists x', x''(x = [x'|x''])$ and $nil(x) \vdash_{\mathcal{A}} x = []$. Roughly speaking, this guarantees that the chosen domain is accurate w.r.t. the ask processes in the program.

The semantics of the process $P = gen_a(x) \parallel gen_b(y) \parallel append(x,y,z)$ is depicted in Figure 10. Assume that $s = c_1.c_2...c_\kappa \in [\![P]\!]^\alpha$. Let $n \leq \kappa$ and assume that for $i < n$, $c_i \vdash_{\mathcal{A}} go_a = []$ and $c_n \vdash_{\mathcal{A}} stop_a = []$. Since $s \in [\![P]\!]^\alpha$, we know that $s \in [\![gen_a(x)]\!]^\alpha$ and then, we can verify that $c_n \vdash^\alpha \langle x, list(x)\rangle$. Similarly, take $m \leq \kappa$ and assume that for $j < m$, $c_j \vdash_{\mathcal{A}} go_b = []$ and $c_m \vdash_{\mathcal{A}} stop_b = []$. We can verify that $c_m \vdash^\alpha \langle y, list(y)\rangle$. Finally, since $s \in append(x,y,z)$, we can show that $c_{max(n,m)} \vdash^\alpha \langle z, list(z)\rangle$. In words, the process P binds x, y and z to ground lists whenever the environment provides as input a series of constraints $go_a = []$ (resp. $go_b = []$) followed by an input $stop_a = []$ (resp. $stop_b = []$).

5.2.1 Reactive Systems

Synchronous data flow languages (Berry and Gonthier 1992) such as Esterel and Lustre can be encoded as `tcc` processes (Saraswat et al. 1994; Tini 1999). This makes `tcc` an expressive declarative framework for the modeling and verification of reactive systems. Take for instance the program in Figure 11, taken and slightly

$$
\begin{array}{l}
micCtrl(Error, Signal) :- \\
\quad (\mathbf{local}\, Error', Signal', er, sl)\, (\\
\qquad !\,\mathbf{tell}(Error = [er \mid Error'] \sqcup Signal = [sl \mid Signal']) \\
\qquad \parallel \mathbf{when}\ \texttt{on} \sqcup \texttt{open}\ \mathbf{do}\ !\,\mathbf{tell}(er = yes \sqcup Error' = [] \sqcup sl = stop) \\
\qquad \parallel \mathbf{when}\ \texttt{off}\ \mathbf{do}\ (!\,\mathbf{tell}(er = no) \parallel \mathbf{next}\, micCtrl(Error', Signal')) \\
\qquad \parallel \mathbf{when}\ \texttt{closed}\ \mathbf{do}\ (!\,\mathbf{tell}(er = no) \parallel \mathbf{next}\, micCtrl(Error', Signal')))
\end{array}
$$

Fig. 11: Model for a microwave controller (see Notation 3 for the definition of !).

modified from (Falaschi and Villanueva 2006), that models a control system for a microwave checking that the door must be closed when it is turned on. Otherwise, it must emit an error signal. In this model, `on`, `off`, `closed` and `open` represent the constraints $on = [], off = [], close = []$ and $open = []$ and the symbols yes, no, $stop$ denote constant symbols.

The analyses developed here can provide additional reasoning techniques in `tcc` for the verification of such systems. For instance, by using the groundness analysis in the previous section, we can show that if $c_1.c_2....c_\kappa \in [\![micCtrl(Error, Button)]\!]^\alpha$ and there exists $1 \leq i \leq \kappa$ s.t. $c_i \vdash_{\mathcal{A}} (open = [] \sqcup on = [])$, then, it must be the case that $c_1 \vdash^\alpha \langle Error, list(Error)\rangle$, i.e., $Error$ is a ground variable. This means, that the system correctly binds the list $Error$ to a ground term whenever the system reaches an inconsistent state.

Observation 3 (*Synchronization constraints*)
In several applications of `tcc` and `utcc` the environment interact with the system by adding as input some constraints that only appear in the guard of ask processes as `on`, `off`, `open`, `close` in Figure 11 and go_a, $stop_a$ in the Figure 9. These constraints can be thought of as "synchronization constraints" (Fages et al. 2001). Furthermore, since these constraints are inputs from the environment, they are not expected to be produced by the program, i.e., they do not appear in the scope of a tell process. In these situations, in order to improve the accuracy of the analyses, one can orthogonally add those constraints in the abstract domain. This can be done, for instance, with a reduced product as we did in Definition 19 to give a finer approximation of the inputs go_a and $stop_a$ by adding type dependency information.

5.3 Suspension Analysis

In a concurrent setting it is important to know whether a given system reaches a state where no further evolution is possible. Reaching a deadlocked situation is something to be avoided. There are many studies on this problem and several works developing analyses in (logic) concurrent languages (e.g. (Codish et al. 1994; Codish et al. 1997)). However, we are not aware of studies available for `ccp` and its temporal extensions. A suspended state in the context of `ccp` may happen when the guard of the ask processes are not carefully chosen and then, none of them can be entailed. In this section we develop an analysis that aims at determining the constraints that a program needs as input from the environment to proceed. This can be used to derive information about the suspension of the system. We start by extending

the concrete semantics to a collecting semantics that keeps information about the suspension of processes. For this, we define the following constraint system.

Definition 20 (*Suspension Constraint System*)
Let $\mathcal{S} = \{\bot, \mathtt{ns}\}$ s.t. $\bot \leq \mathtt{ns}$. Given a constraint system $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \mathtt{t}, \mathtt{f}, Var, \exists, d\rangle$, the suspension-constraint system $S(\mathbf{C})$ is defined as

$$\mathbf{S} = \langle \mathcal{C} \times \mathcal{S}, \leq^s, \sqcup^s, \langle \mathtt{t}, \bot\rangle, \langle \mathtt{f}, \mathtt{ns}\rangle, Var, \exists^s, d^s\rangle$$

where $\leq^s, \sqcup^s$ are pointwise defined, $\exists^s_{\vec{x}}(\langle c, c'\rangle) = \langle \exists_x c, c'\rangle$ and $d^s_{\vec{x}\vec{t}} = \langle d_{\vec{x}\vec{t}}, \bot\rangle$. Given a constraint $c \in \mathcal{C}$, we shall use $\widehat{c}$ to denote the constraint $\langle c, \bot\rangle$.

Let us illustrate how $S(\mathbf{C})$ allows us to derive information about suspension.

Example 10 (*Collecting Semantics*)
Let $\mathcal{C} = \{\mathtt{t}, a, b, c, d, \mathtt{f}\}$ be a complete lattice where $b \vdash a$ and $d \vdash c$, $P = \mathbf{when}\ a\ \mathbf{do}\ \mathbf{tell}(b)$ and $Q = \mathbf{when}\ c\ \mathbf{do}\ \mathbf{tell}(d)$. We know that $[\![P]\!] = \{\mathtt{t}, b, c, d, \mathtt{f}\}.\mathcal{C}^\omega$ (note that P does not suspend on b and $\mathtt{f}$). Let $\widehat{P}$ and $\widehat{Q}$ be defined over $S(\mathbf{C})$ as:

$$\widehat{P} = \mathbf{when}\ \widehat{a}\ \mathbf{do}\ (\mathbf{tell}(\widehat{b}) \parallel \mathbf{tell}(\langle a, \mathtt{ns}\rangle)) \qquad \widehat{Q} = \mathbf{when}\ \widehat{c}\ \mathbf{do}\ (\mathbf{tell}(\widehat{d}) \parallel \mathbf{tell}(\langle c, \mathtt{ns}\rangle))$$

We then have:

$$\begin{array}{rcl} [\![\widehat{P}]\!] & = & \{\langle \mathtt{t}, \uparrow\bot\rangle, \langle b, \mathtt{ns}\rangle, \langle c, \uparrow\bot\rangle, \langle d, \uparrow\bot\rangle, \langle \mathtt{f}, \mathtt{ns}\rangle\}.(\mathcal{C} \times \mathcal{S})^\omega \\ [\![\widehat{Q}]\!] & = & \{\langle \mathtt{t}, \uparrow\bot\rangle, \langle a, \uparrow\bot\rangle, \langle b, \uparrow\bot\rangle, \langle d, \mathtt{ns}\rangle, \langle \mathtt{f}, \mathtt{ns}\rangle\}.(\mathcal{C} \times \mathcal{S})^\omega \\ [\![\widehat{P} \parallel \widehat{Q}]\!] & = & \{\langle \mathtt{t}, \uparrow\bot\rangle, \langle b, \mathtt{ns}\rangle, \langle d, \mathtt{ns}\rangle, \langle \mathtt{f}, \mathtt{ns}\rangle\}.(\mathcal{C} \times \mathcal{S})^\omega \end{array}$$

where $\langle c, \uparrow\bot\rangle$ is a shorthand for the couple of tuples $\langle c, \bot\rangle, \langle c, \mathtt{ns}\rangle$. The process P suspends on input c (since $c \not\vdash a$) while Q under input c outputs d and it does not suspend. Notice that the system $P \parallel Q$ does not block on input b, d or $\mathtt{f}$ and it does on input $\mathtt{t}$. Notice also that $\langle c, \bot\rangle.s \notin [\![\widehat{P} \parallel \widehat{Q}]\!]$. This means that in a store c, at least one the ask processes in $\widehat{P} \parallel \widehat{Q}$ is able to proceed. The key idea is that the process $\mathbf{tell}(\langle c, \mathtt{ns}\rangle)$ in $\widehat{Q}$ ensures that if $\langle e, e'\rangle \in [\![\widehat{Q}]\!]$ and $e \vdash c$, then it must be the case that $e' = \mathtt{ns}$. This corresponds to the intuition that if an ask process can evolve on a store c, it can evolve under any store greater than c (Lemma 1).

Next we define a program transformation that allows us to scatter suspension information when we want to verify that none of the ask processes suspend.

Example 11
Let P and Q be as in Example 10. Let also $\widehat{P} = \mathbf{when}\ \widehat{a}\ \mathbf{do}\ (\mathbf{tell}(\widehat{b}))$, $\widehat{Q} = \mathbf{when}\ \widehat{c}\ \mathbf{do}\ (\mathbf{tell}(\widehat{d}))$ and $\widehat{R} = \widehat{P} \parallel \widehat{Q} \parallel \mathbf{when}\ \widehat{a} \sqcup \widehat{c}\ \mathbf{do}\ (\mathbf{tell}(a \sqcup c, \mathtt{ns}))$. Therefore,

$$\begin{array}{rcl} [\![\widehat{P}]\!] & = & \{\langle \mathtt{t}, \uparrow\bot\rangle, \langle b, \uparrow\bot\rangle, \langle c, \uparrow\bot\rangle, \langle d, \uparrow\bot\rangle, \langle \mathtt{f}, \uparrow\bot\rangle\}.(\mathcal{C} \times \mathcal{S})^\omega \\ [\![\widehat{Q}]\!] & = & \{\langle \mathtt{t}, \uparrow\bot\rangle, \langle a, \uparrow\bot\rangle, \langle b, \uparrow\bot\rangle, \langle d, \uparrow\bot\rangle, \langle \mathtt{f}, \uparrow\bot\rangle\}.(\mathcal{C} \times \mathcal{S})^\omega \\ [\![\widehat{R}]\!] & = & \{\langle \mathtt{t}, \uparrow\bot\rangle, \langle b, \uparrow\bot\rangle, \langle d, \uparrow\bot\rangle, \langle \mathtt{f}, \mathtt{ns}\rangle\}.(\mathcal{C} \times \mathcal{S})^\omega \end{array}$$

Hence we can conclude that only under input $\mathtt{f}$ neither P nor Q suspend.

The previous program transformation can be arbitrarily applied to subterms of the form $P = \prod_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$. Similarly, for verification purposes, a subterm of the form $P = (\mathbf{abs}\ \vec{x_1}; c_1)\, P_1 \parallel ... \parallel (\mathbf{abs}\ \vec{x_n}; c_n)\, P_n$ can be replaced by

$$P' = \widehat{P} \parallel \mathbf{when}\ (\exists \vec{x_1} \widehat{c_1} \sqcup ... \sqcup \exists \vec{x_n} \widehat{c_n})\ \mathbf{do}\ \mathbf{tell}(\langle c_1 \sqcup ... \sqcup c_n, \mathtt{ns}\rangle)$$

$$
\begin{array}{rcl}
[\![Protocol]\!]^\alpha & = & A.\epsilon \cap S.\epsilon \cap B.\epsilon \text{ where} \\
A & = & \exists\, m(\uparrow(\widehat{\mathtt{out}}(\{x,y,m\}_{pub(srv)}))) \\
S & = & \forall\, x,y,m(\{\langle d,c\rangle \mid \text{ if } \langle d,c\rangle \vdash_{\mathcal{A}} \widehat{\mathtt{out}}(\{x,y,m\}_{pub(srv)}) \\
 & & \qquad\qquad \text{then } \langle d,c\rangle \leq^\alpha \widehat{\mathtt{out}}(\{x,m\}_{pub(y)})\}) \\
B & = & \forall\, x,m(\{\langle d,c\rangle \mid \text{ if } \langle d,c\rangle \vdash_{\mathcal{A}} \widehat{\mathtt{out}}(\{x,m\}_{pub(y)}) \\
 & & \qquad\qquad \text{then } \langle d,c\rangle \leq^\alpha \langle \mathtt{out}(\{x,m\}_{pub(y)}), \mathtt{ns}\rangle\})\ \}
\end{array}
$$

Fig. 12: Semantics of the protocol in Example 12.

We conclude with an example showing how an abstraction of the previous collecting semantics allows us to analyze a protocol programmed in `utcc`. For this we shall use the abstraction in Definition 18 to cut the terms up to a given length.

Example 12
Assume a protocol where agent A has to send a message to B through a proxy server S. This situation can be modeled as follows:

$$
\begin{array}{lcl}
A(x,y) & :- & (\mathbf{local}\, m)\, (\mathbf{tell}(\mathtt{out}(\{x,y,m\}_{pub(srv)}))) \\
S & :- & (\mathbf{abs}\ x,y,m; \mathtt{out}(\{x,y,m\}_{pub(srv)}))\, \mathbf{tell}(\mathtt{out}(\{x,m\}_{pub(y)}))\, \|\, \mathbf{next}\, S() \\
B(y) & :- & (\mathbf{abs}\ x,m; \mathtt{out}(\{x,m\}_{pub(y)}))\, B_c \\
Protocol & :- & A(x,y) \parallel S() \parallel B(y)
\end{array}
$$

where $B_c = \mathbf{skip}$ is the continuation of the protocol that we left unspecified.

This code is correct if the message can flow from A to B without any input from the environment. This holds if the ask process in $B(y)$ does not block. We shall then analyze the program above by replacing all c with $\widehat{c}$ and $B(y)$ with

$$B'(y) :- (\mathbf{abs}\ x,m; \widehat{\mathtt{out}}(\{x,m\}_{pub(y)}))\, (\mathbf{tell}(\langle \mathtt{out}(\{x,m\}_{pub(y)}), \mathtt{ns}\rangle))$$

Let α_κ be as in Definition 18. We choose as abstract domain $\mathcal{A} = S(\alpha_\kappa(\mathcal{C}))$ and we consider sequences of length one. In Figure 12 we show the abstract semantics. We notice that $\langle c, \mathtt{ns}\rangle$ where $c = \exists m(\mathtt{out}(\{x,y,m\}_{pub(srv)}) \sqcup \mathtt{out}(\{x,m\}_{pub(y)}))$ is in the semantics $[\![Protocol]\!]^\alpha$ and $\langle c, \bot\rangle \notin [\![Protocol]\!]^\alpha$. We then conclude that the protocol is able to correctly deliver the message to B.

Assume now that the code for the server is (wrongly) written as

$$S' :- (\mathbf{abs}\ x,y,m; \mathtt{out}(\{x,y,m\}_{pub(srv)}))\, \mathbf{tell}(\mathtt{out}(\{x,m\}_{pub(x)})) \parallel \mathbf{next}\, S'()$$

where we changed $\mathbf{tell}(\mathtt{out}(\{x,m\}_{pub(y)}))$ to $\mathbf{tell}(\mathtt{out}(\{x,m\}_{pub(x)}))$. We can verify that $\langle c, \bot\rangle \in [\![Protocol'\!]\!]^\alpha$ where $c = \exists m(\mathtt{out}(\{x,y,m\}_{pub(srv)}) \sqcup \mathtt{out}(\{x,m\}_{pub(x)}))$. This can warn the programmer that there is a mistake in the code.

6 Concluding Remarks

Several frameworks and abstract domains for the analysis of logic programs have been defined (see e.g. (Cousot and Cousot 1992; Codish et al. 1999; Armstrong et al. 1998)). Those works differ from ours since they do not deal with the temporal behavior and synchronization mechanisms present in `tcc`-based languages. On the contrary, since our framework is parametric w.r.t. the abstract domain, it can benefit from those works.

We defined in (Falaschi et al. 2007) a framework for the declarative debugging of `ntcc` (Nielsen et al. 2002a) programs (a non-deterministic extension of `tcc`). The framework presented here is more general since it was designed for the static analysis of `tcc` and `utcc` programs and not only for debugging. Furthermore, as mentioned above, it is parametric w.r.t an abstract domain. In (Falaschi et al. 2007) we also dealt with infinite sequences of constraints and a similar finite cut over sequences was proposed there.

In (Olarte and Valencia 2008b) a symbolic semantics for `utcc` was proposed to deal with the infinite internal reductions of non well-terminated processes. This semantics, by means of temporal formulas, represents finitely the infinitely many constraints (and substitutions) the SOS may produce. The work in (Olarte and Valencia 2008a) introduces a denotational semantics for `utcc` based on (partial) closure operators over sequences of *temporal logic formulas*. This semantics captures compositionally the *symbolic strongest postcondition* and it was shown to be fully abstract w.r.t. the symbolic semantics for the fragment of locally-independent (see Definition 10) and abstracted-unless free processes (i.e., processes not containing occurrences of **unless** processes in the scope of abstractions). The semantics here presented turns out to be more appropriate to develop the abstract interpretation framework in Section 4. Firstly, the inclusion relation between the strongest postcondition and the semantics is verified for the whole language (Theorem 3) – in (Olarte and Valencia 2008a) this inclusion is verified only for the abstracted-unless free fragment–. Secondly, this semantics makes use of the entailment relation over constraints rather than the more involved entailment over first-order linear-time temporal formulas as in (Olarte and Valencia 2008a). Finally, our semantics allows us to capture the behavior of `tcc` programs with recursion. This is not possible with the semantics in (Olarte and Valencia 2008a) which was thought only for `utcc` programs where recursion can be encoded. This work then provides the theoretical basis for building tools for the data-flow analyses of `utcc` and `tcc` programs.

For the kind of applications that stimulated the development of `utcc`, it was defined entirely deterministic. The semantics here presented could smoothly be extended to deal with some forms of non-determinism like those in (Falaschi et al. 1997a), thus widening the spectrum of applications of our framework.

A framework for the abstract diagnosis of timed-concurrent constraint programs has been defined in (Comini et al. 2011) where the authors consider a denotational semantics similar to ours, although with several technical differences. The language studied in (Comini et al. 2011) corresponds to `tccp` (de Boer et al. 2000), a temporal `ccp` language where the stores are monotonically accumulated along the time-units and whose operational semantics relies on the notion of true parallelism. We note that the framework developed in (Comini et al. 2011) is used for abstract diagnosis rather than for general analyses.

Our results should foster the development of analyzers for different systems modeled in `utcc` and its sub-calculi such as security protocols, reactive and timed systems, biological systems, etc (see (Olarte et al. 2013) for a survey of applications of `ccp`-based languages). We plan also to perform freeness, suspension, type and independence analyses among others. It is well known that this kind of analyses have

many applications, e.g. for code optimization in compilers, for improving run-time execution, and for approximated verification. We also plan to use abstract model checking techniques based on the proposed semantics to automatically analyze `utcc` and `tcc` code.

Acknowledgments. We thank Frank D. Valencia, François Fages and Rémy Haemmerlé for insightful discussions on different subjects related to this work. We also thank the anonymous reviewers for their detailed comments. Special thanks to Emanuele D'Osualdo for his careful remarks and suggestions for improving the paper. This work has been partially supported by grant 1251-521-28471 from Colciencias, and by Digiteo and DGAR funds for visitors.

References

Armando, A. and Compagna, L. 2008. Sat-based model-checking for security protocols analysis. *Internation Journal of Information Security 7,* 1, 3–32.

Armstrong, T., Marriott, K., Schachte, P., and Søndergaard, H. 1998. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming 31,* 1, 3–45.

Berry, G. and Gonthier, G. 1992. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming 19,* 2, 87–152.

Bodei, C., Brodo, L., Degano, P., and Gao, H. 2010. Detecting and preventing type flaws at static time. *Journal of Computer Security 18,* 2, 229–264.

Boreale, M. 2001. Symbolic trace analysis of cryptographic protocols. In *ICALP*, F. Orejas, P. G. Spirakis, and J. van Leeuwen, Eds. LNCS, vol. 2076. Springer, 667–681.

Codish, M. and Demoen, B. 1994. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *SAS*, B. L. Charlier, Ed. LNCS, vol. 864. Springer, 281–296.

Codish, M., Falaschi, M., and Marriott, K. 1994. Suspension Analyses for Concurrent Logic Programs. *ACM Transactions on Programming Languages and Systems 16,* 3, 649–686.

Codish, M., Falaschi, M., Marriott, K., and Winsborough, W. 1997. A Confluent Semantic Basis for the Analysis of Concurrent Constraint Logic Programs. *Journal of Logic Programming 30,* 1, 53–81.

Codish, M., Søndergaard, H., and Stuckey, P. 1999. Sharing and groundness dependencies in logic programs. *ACM Transations on Programming Languages and Systems 21,* 5, 948–976.

Comini, M., Titolo, L., and Villanueva, A. 2011. Abstract diagnosis for timed concurrent constraint programs. *TPLP 11,* 4-5, 487–502.

Cousot, P. and Cousot, R. 1979. Systematic design of program analysis frameworks. In *POPL*, A. V. Aho, S. N. Zilles, and B. K. Rosen, Eds. ACM Press, 269–282.

Cousot, P. and Cousot, R. 1992. Abstract Interpretation and Applications to Logic Programs. *Journal of Logic Programming 13,* 2&3, 103–179.

de Boer, F. S., Gabbrielli, M., Marchiori, E., and Palamidessi, C. 1997. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems 19,* 5, 685–725.

de Boer, F. S., Gabbrielli, M., and Meo, M. C. 2000. A timed concurrent constraint language. *Inf. Comput. 161,* 1, 45–83.

de Boer, F. S., Pierro, A. D., and Palamidessi, C. 1995. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science 151,* 1, 37–78.

Dolev, D. and Yao, A. C. 1983. On the security of public key protocols. *IEEE Transactions on Information Theory 29,* 12, 198–208.

Escobar, S., Meadows, C., and Meseguer, J. 2011. State space reduction in the maude-nrl protocol analyzer. *CoRR abs/1105.5282.*

Fages, F., Ruet, P., and Soliman, S. 2001. Linear concurrent constraint programming: Operational and phase semantics. *Inf. Comput. 165,* 1, 14–41.

Falaschi, M., Gabbrielli, M., Marriott, K., and Palamidessi, C. 1993. Compositional analysis for concurrent constraint programming. In *LICS*. IEEE Computer Society, 210–221.

Falaschi, M., Gabbrielli, M., Marriott, K., and Palamidessi, C. 1997a. Confluence in concurrent constraint programming. *Theoretical Computer Science 183,* 2, 281–315.

Falaschi, M., Gabbrielli, M., Marriott, K., and Palamidessi, C. 1997b. Constraint logic programming with dynamic scheduling: A semantics based on closure operators. *Inf. Comput. 137,* 1, 41–67.

Falaschi, M., Olarte, C., and Palamidessi, C. 2009. A framework for abstract interpretation of timed concurrent constraint programs. In *PPDP*, A. Porto and F. J. López-Fraguas, Eds. ACM, 207–218.

Falaschi, M., Olarte, C., Palamidessi, C., and Valencia, F. 2007. Declarative diagnosis of temporal concurrent constraint programs. In *ICLP*, V. Dahl and I. Niemelä, Eds. LNCS, vol. 4670. Springer, 271–285.

Falaschi, M. and Villanueva, A. 2006. Automatic verification of timed concurrent constraint programs. *TPLP 6,* 3, 265–300.

Fiore, M. P. and Abadi, M. 2001. Computing symbolic models for verifying cryptographic protocols. In *CSFW*. IEEE Computer Society, 160–173.

Giacobazzi, R., Debray, S. K., and Levi, G. 1995. Generalized semantics and abstract interpretation for constraint logic programs. *J. Log. Program. 25,* 3, 191–247.

Haemmerlé, R., Fages, F., and Soliman, S. 2007. Closures and modules within linear logic concurrent constraint programming. In *FSTTCS*, V. Arvind and S. Prasad, Eds. LNCS, vol. 4855. Springer, 544–556.

Hentenryck, P. V., Saraswat, V. A., and Deville, Y. 1998. Design, implementation, and evaluation of the constraint language cc(fd). *Journal of Logic Programming 37,* 1-3, 139–164.

Hildebrandt, T. and López, H. A. 2009. Types for secure pattern matching with local knowledge in universal concurrent constraint programming. In *ICLP*, P. M. Hill and D. S. Warren, Eds. LNCS, vol. 5649. Springer, 417–431.

Jaffar, J. and Lassez, J.-L. 1987. Constraint logic programming. In *POPL*. ACM Press, 111–119.

Jagadeesan, R., Marrero, W., Pitcher, C., and Saraswat, V. A. 2005. Timed constraint programming: a declarative approach to usage control. In *PPDP*, P. Barahona and A. P. Felty, Eds. ACM, 164–175.

López, H. A., Olarte, C., and Pérez, J. A. 2009. Towards a unified framework for declarative structured communications. In *PLACES*, A. R. Beresford and S. J. Gay, Eds. EPTCS, vol. 17. 1–15.

Lowe, G. 1996. Breaking and fixing the needham-schroeder public-key protocol using fdr. *Software - Concepts and Tools 17,* 3, 93–102.

Maher, M. J. 1988. Complete axiomatizations of the algebras of finite, rational and infinite trees. In *LICS*. IEEE Computer Society, 348–357.

Mendler, N. P., Panangaden, P., Scott, P. J., and Seely, R. A. G. 1995. A logical view of concurrent constraint programming. *Nordic Journal of Computing 2,* 2, 181–220.

Milner, R., Parrow, J., and Walker, D. 1992. A calculus of mobile processes, Parts I and II. *Inf. Comput. 100,* 1, 1–40.

Nielsen, M., Palamidessi, C., and Valencia, F. 2002a. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic J. of Computing 9,* 1, 145–188.

Nielsen, M., Palamidessi, C., and Valencia, F. D. 2002b. On the expressive power of temporal concurrent constraint programming languages. In *PPDP*. ACM, 156–167.

Olarte, C., Rueda, C., and Valencia, F. D. 2013. Models and emerging trends of concurrent constraint programming. *Constraints 18,* 4, 535–578.

Olarte, C. and Valencia, F. D. 2008a. The expressivity of universal timed CCP: undecidability of monadic FLTL and closure operators for security. In *PPDP*, S. Antoy and E. Albert, Eds. ACM, 8–19.

Olarte, C. and Valencia, F. D. 2008b. Universal concurrent constraint programing: symbolic semantics and applications to security. In *SAC*, R. L. Wainwright and H. Haddad, Eds. ACM, 145–150.

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

Saraswat, V. A., Jagadeesan, R., and Gupta, V. 1994. Foundations of timed concurrent constraint programming. In *LICS*. IEEE Computer Society, 71–80.

Saraswat, V. A., Rinard, M. C., and Panangaden, P. 1991. Semantic foundations of concurrent constraint programming. In *POPL*, D. S. Wise, Ed. ACM Press, 333–352.

Sato, T. and Tamaki, H. 1984. Enumeration of Success Patterns in Logic Programs. *Theoretical Computer Science 34*, 227–240.

Shapiro, E. Y. 1989. The family of concurrent logic programming languages. *ACM Comput. Surv. 21,* 3, 413–510.

Smolka, G. 1994. A foundation for higher-order concurrent constraint programming. In *CCL*, J.-P. Jouannaud, Ed. LNCS, vol. 845. Springer, 50–72.

Song, D. X., Berezin, S., and Perrig, A. 2001. Athena: A novel approach to efficient automatic security protocol analysis. *Journal of Computer Security 9,* 1/2, 47–74.

Tini, S. 1999. On the expressiveness of timed concurrent constraint programming. *Electr. Notes Theor. Comput. Sci. 27*, 3–17.

Zaffanella, E., Giacobazzi, R., and Levi, G. 1997. Abstracting synchronization in concurrent constraint programming. *Journal of Functional and Logic Programming 1997,* 6.

Appendix A Detailed proofs Section 2.4

Before presenting the proof that `utcc` is deterministic, we shall prove the following auxiliary result.

Lemma 6 (*Confluence*)
Suppose that $\gamma_0 \longrightarrow \gamma_1$, $\gamma_0 \longrightarrow \gamma_2$ and $\gamma_1 \not\equiv \gamma_2$. Then, there exists γ_3 such that $\gamma_1 \longrightarrow \gamma_3$ and $\gamma_2 \longrightarrow \gamma_3$.

Proof
Let $\gamma_0 = \langle \vec{x}; P; c\rangle$. The proof proceed by structural induction on P. In each case where γ_0 has two different transitions (up to $\equiv$) $\gamma_0 \longrightarrow \gamma_1$ and $\gamma_0 \longrightarrow \gamma_2$, one shows the existence of γ_3 s.t. $\gamma_1 \longrightarrow \gamma_3$ and $\gamma_2 \longrightarrow \gamma_3$.

Given a configuration $\gamma = \langle \vec{x}; P; c\rangle$ let us define the size of γ as the size of P as follows: $M(\mathbf{skip}) = 0$, $M(\mathbf{tell}(c)) = M(p(\vec{t})) = 1$, $M((\mathbf{abs}\ \vec{x}; c; D)\, P') = M((\mathbf{local}\, \vec{x})\, P') = M(\mathbf{next}\, P') = M(\mathbf{unless}\ c\ \mathbf{next}\, P') = 1 + M(P')$ and $M(Q \parallel R) = M(Q) + M(R)$. Suppose that $\gamma_0 \equiv \langle \vec{x}; P; c_0\rangle$, $\gamma_0 \longrightarrow \gamma_1$, $\gamma_0 \longrightarrow \gamma_2$ and $\gamma_1 \not\equiv \gamma_2$. The proof proceeds by induction on the size of γ_0. From the assumption $\gamma_1 \not\equiv \gamma_2$, it must be the case that the transition $\longrightarrow$ is not an instance of the rule $\mathrm{R}_{\mathrm{STRVAR}}$; moreover, P is neither a process of the form $\mathbf{tell}(c)$, $(\mathbf{local}\, \vec{x})\, P$, $p(\vec{t})$ or $\mathbf{unless}\ c\ \mathbf{next}\, P'$ (since those processes have a unique possible transition modulo structural congruence) nor $\mathbf{next}\, P$ or $\mathbf{skip}$ (since they do not exhibit any internal derivation).

For the case $P = Q \parallel R$, we have to consider three cases. Assume that $\gamma_1 \equiv \langle \vec{x}_1; Q_1 \parallel R, c_1\rangle$ and $\gamma_2 \equiv \langle \vec{x}_2; Q_2 \parallel R, c_2\rangle$. Let $\gamma_0' \equiv \langle \vec{x}; Q; c_0\rangle$, $\gamma_1' \equiv \langle \vec{x}_1; Q_1; c_1\rangle$ and $\gamma_2' \equiv \langle \vec{x}_2; Q_2; c_2\rangle$. We know by induction that if $\gamma_0' \longrightarrow \gamma_1'$ and $\gamma_0' \longrightarrow \gamma_2'$ then there exists $\gamma_3' \equiv \langle \vec{x}_3; Q_3; c_3\rangle$ such that $\gamma_1' \longrightarrow \gamma_3'$ and $\gamma_2' \longrightarrow \gamma_3'$. We conclude by noticing that $\gamma_1 \longrightarrow \gamma_3$ and $\gamma_2 \longrightarrow \gamma_3$ where $\gamma_3 \equiv \langle \vec{x}_3; Q_3 \parallel R; c_3\rangle$. The remaining cases when (1) R has two possible transitions and (2) when Q moves to Q' and then R moves to R' are similar.

Let $\gamma_0 \equiv \langle \vec{x}; P; c_0\rangle$ with $P = (\mathbf{abs}\ \vec{y}; c; D)\, Q$. One can verify that $\gamma_1 \equiv \langle \vec{x} \cup \vec{x}_1; P_1; c_0\rangle$ where P_1 takes the form $(\mathbf{abs}\ \vec{y}; c; D \cup \{d_{\vec{y}\vec{t_1}}\})\, Q \parallel Q[\vec{t_1}/\vec{y}]$ and $\gamma_2 \equiv \langle \vec{x} \cup \vec{x}_2; P_2; c_0\rangle$ where P_2 takes the form $(\mathbf{abs}\ \vec{z}; c; D \cup \{d_{\vec{y}\vec{t_2}}\})\, Q \parallel Q[\vec{t_2}/\vec{y}]$. From the assumption $\gamma_1 \not\equiv \gamma_2$, it must be the case that $d_{\vec{y}\vec{t_1}} \ncong d_{\vec{y}\vec{t_2}}$. By alpha conversion we assume that $\vec{x}_1 \cap \vec{x}_2 = \emptyset$. Let $\gamma_3 \equiv \langle \vec{x} \cup \vec{x}_1 \cup \vec{x}_2; P_3; c_0\rangle$ where $P_3 = (\mathbf{abs}\ \vec{y}; c; D \cup \{d_{\vec{y}\vec{t_1}}, d_{\vec{y}\vec{t_2}}\})\, Q \parallel Q[\vec{t_1}/\vec{y}] \parallel Q[\vec{t_2}/\vec{y}]$. Clearly $\gamma_1 \longrightarrow \gamma_3$ and $\gamma_2 \longrightarrow \gamma_3$ as wanted. □

Observation 4 (*Finite Traces*)
Let $\gamma_1 \longrightarrow \cdots \longrightarrow \gamma_n \not\longrightarrow$ by a finite internal derivation. The number of possible internal transitions (up to $\equiv$) in any $\gamma_i = \langle \vec{x}_i; P_i; c_i\rangle$ in the above derivation is finite.

Proof
We proceed by structural induction on P_i. The interesting case is the **abs** process. Let $Q = (\mathbf{abs}\ \vec{x}; c)\, P$. Suppose, to obtain a contradiction, that $c_i \vdash c[\vec{t}/\vec{x}]$ for infinitely many $\vec{t}$ (to have infinitely many possible internal transitions). In that case, it is easy to see that we must have infinitely many internal derivation, thus contradicting the assumption that $\gamma_n \not\longrightarrow$. □

Lemma 7 (*Finite Traces*)
If there is a finite internal derivation of the form $\gamma_1 \longrightarrow \gamma_2 \longrightarrow \cdots \longrightarrow \gamma_n \not\longrightarrow$ then, any derivation starting from γ_1 is finite.

Proof
We observe that recursive calls must be guarded by a **next** processes. Then, any infinite behavior inside a time-unit is due to an **abs** process. From Observation 4 and Lemma 6, it follows that any derivation starting from γ_1 is finite. □

Theorem 1 (*Determinism*)
Let s, w and w' be (possibly infinite) sequences of constraints. If both (s, w), $(s, w') \in io(P)$ then $w \cong w'$.

Proof
Assume that $P \stackrel{(c,\exists\vec{x}(d))}{\Longrightarrow} (\mathbf{local}\,\vec{x})\,F(Q)$, $P \stackrel{(c,\exists\vec{x}'(d'))}{\Longrightarrow} (\mathbf{local}\,\vec{x}')\,F(Q')$ and let $\gamma_1 \equiv \langle\emptyset; P; c\rangle$, $\gamma_2 \equiv \langle\emptyset; P; c\rangle$. If $\gamma_1 \not\longrightarrow$ then trivially $\gamma_2 \not\longrightarrow$, $d \cong d'$ and $Q \equiv Q'$. Now assume that $\gamma_1 \longrightarrow^* \gamma_1' \not\longrightarrow$ and $\gamma_2 \longrightarrow^* \gamma_2' \not\longrightarrow$ where $\gamma_1' \equiv \langle\vec{x}; Q; d\rangle$ and $\gamma_2' \equiv \langle\vec{x}'; Q'; d'\rangle$. By repeated applications of Lemma 6 we conclude $\gamma_1' \equiv \gamma_2'$ and then, $d \cong d'$ and $Q \equiv Q'$. □

Lemma 2 (*Closure Properties*)
Let P be a process. Then,

(1) $io(P)$ is a function.
(2) $io(P)$ is a partial closure operator, namely it satisfies:
Extensiveness: If $(s, s') \in io(P)$ then $s \leq s'$.
Idempotence: If $(s, s') \in io(P)$ then $(s', s') \in io(P)$.
Monotonicity: Let P be a monotonic process such that $(s_1, s_1') \in io(P)$. If $(s_2, s_2') \in io(P)$ and $s_1 \leq s_2$, then $s_1' \leq s_2'$.

Proof
We shall assume here that the input and output sequences are infinite. The proof for the case when the sequences are finite is analogous. The proof of (1) is immediate from Theorem 1. For (2), assume that $s = c_1.c_2...$, $s' = c_1'.c_2'...$ and that $(s, s') \in io(P)$. We then have a derivation of the form:

$$P \equiv P_1 \stackrel{(c_1,c_1')}{\Longrightarrow} P_2 \stackrel{(c_2,c_2')}{\Longrightarrow} ...P_i \stackrel{(c_i,c_i')}{\Longrightarrow} P_{i+1}...$$

For $i \geq 1$, we also know that there is an internal derivation of the form $\langle\emptyset; P_i; c_i\rangle \longrightarrow^* \langle\vec{x}; P_i'; c_i'\rangle \not\longrightarrow$ where $P_{i+1} = (\mathbf{local}\,\vec{x})\,F(P_i')$.
Extensiveness follows from (1) in Lemma 1.
Idempotence is proved by repeated applications of (3) in Lemma 1.

As for **Monotonicity**, we proceed as in (Nielsen et al. 2002a). Let $\preceq$ be the minimal ordering relation on processes satisfying: (1) $\mathbf{skip} \preceq P$. (2) If $P \preceq Q$ and $P \equiv P'$ and $Q \equiv Q'$ then $P' \preceq Q'$. (3) If $P \preceq Q$, for every context $C[\cdot]$, $C[P] \preceq C[Q]$. Intuitively, $P \preceq Q$ represents the fact that Q contains "at least as much code" as P. We have to show that for every P, P', c, c' and $\vec{x}, \vec{x}'$ if $\langle\vec{x}; P; c\rangle \longrightarrow^* \langle\vec{x}'; P'; c'\rangle \not\longrightarrow$

then for every $d \vdash c$ and Q s.t. $P \preceq Q$ there $\langle \vec{x}; Q; d\rangle \longrightarrow^* \langle \vec{y}; Q'; d'\rangle \not\longrightarrow$ for some $\vec{y}$ and Q' with $(\mathbf{local}\,\vec{x}')\,F(P') \preceq (\mathbf{local}\,\vec{y})\,F(Q')$ and $\exists\vec{y}(d') \vdash \exists\vec{x}'(c')$. This can be proved by induction on the length of the derivation using the following two properties:
(**a**) $\longrightarrow$ is monotonic w.r.t. the store, in the sense that, if $\langle \vec{x}; P; c\rangle \longrightarrow \langle \vec{x}'; P'; c'\rangle$ then for every $d \vdash c$ and Q s.t. $P \preceq Q$, $\langle \vec{x}; Q; d\rangle \longrightarrow \langle \vec{y}; Q'; d'\rangle$ where $\exists\vec{y}(d') \vdash \exists\vec{x}'(c')$ and $(\mathbf{local}\,\vec{x}')\,P' \preceq (\mathbf{local}\,\vec{y})\,Q'$.
(**b**) For every monotonic process P, if $\langle \vec{x}; P; c\rangle \not\longrightarrow$ then for every $d \vdash c$ and Q such that $P \preceq Q$ we have either $\langle \vec{x}; Q; d\rangle \not\longrightarrow$ or $\langle \vec{x}; Q; d\rangle \longrightarrow^* \langle \vec{x}'; Q'; d'\rangle \not\longrightarrow$ where $\exists\vec{x}'(d') \vdash \exists\vec{x}(d)$ and $(\mathbf{local}\,\vec{x})\,F(P) \preceq (\mathbf{local}\,\vec{x}')\,F(Q')$. The restriction to programs which do not contain **unless** constructs is essential here. □

Theorem 2
Let min be the minimum function w.r.t. the order induced by $\leq$ and P be a monotonic process. Then, $(s, s') \in io(P)$ iff $s' = min(sp(P) \cap \{w \mid s \leq w\})$

Proof
Let P be a monotonic process and $(s, s') \in io(P)$. By extensiveness $s \leq s'$ and by idempotence, $(s', s') \in io(P)$. Let $s'' = min(sp(P) \cap \{w \mid s \leq w\})$. Since $s' \in sp(P)$ and $s \leq s'$, it must be the case that $s \leq s'' \leq s'$. If $(s'', s''') \in io(P)$, by monotonicity $s' \leq s'''$. Since $s'' \in sp(P)$, $s'' \cong s'''$ and then, $s' \leq s''$. We conclude $s' \cong s''$. □

Appendix B Detailed Proofs Section 3

Observation 1 (*Equality and $\vec{x}$-variants*)
Let $S \subseteq \mathcal{C}^\omega$, $\vec{z} \subseteq Var$ and s, w be $\vec{x}$-variants such that $d^\omega_{\vec{x}\vec{t}} \leq s$, $d^\omega_{\vec{x}\vec{t}} \leq w$ and $adm(\vec{x}, \vec{t})$. (1) $s \cong w$. (2) $\exists\vec{z}(s) \in \forall\,\vec{x}(S)$ iff $s \in \forall\,\vec{x}(S)$.

Proof
(1) Let $i \geq 1$, $c = s(i)$ and $d = w(i)$. We prove that $c \vdash d$ and $d \vdash c$. We know that $c \sqcup d_{\vec{x}\vec{t}} \cong c$, $d \sqcup d_{\vec{x}\vec{t}} \cong d$ and $\exists\vec{x}(c \sqcup d_{\vec{x}\vec{t}}) \cong \exists\vec{x}(d \sqcup d_{\vec{x}\vec{t}})$. Hence, $c[\vec{t}/\vec{x}] \cong d[\vec{t}/\vec{x}]$. Since $c \vdash \exists\vec{x}(c)$, we can show that $c \vdash \exists\vec{x}(d \sqcup d_{\vec{x}\vec{t}})$ and then, $c \vdash d[\vec{t}/\vec{x}]$. Since $d[\vec{t}/\vec{x}] \sqcup d_{\vec{x}\vec{t}} \vdash d$ (Notation 2) we conclude $c \vdash d$. The "$d \vdash c$" side is analogous and we conclude $c \cong d$.
Property (2) follows directly from the definition of $\forall(\cdot)$. □

Lemma 4
Let $[\![\cdot]\!]$ be as in Definition 9. If $P \xrightarrow{(d,d')} R$ and $d \cong d'$, then $d.[\![R]\!] \subseteq [\![P]\!]$.

Proof
Assume that $\langle \vec{x}; P; d\rangle \longrightarrow^* \langle \vec{x}'; P'; d'\rangle \not\longrightarrow$, $\exists\vec{x}(d) \cong \exists\vec{x}'(d')$. We shall prove that $\exists\vec{x}(d).\exists\,\vec{x}'([\![F(P'))]\!] \subseteq \exists\,\vec{x}([\![P]\!])$. We proceed by induction on the lexicographical order on the length of the internal derivation and the structure of P, where the predominant component is the length of the derivation. Here we present the missing cases in the body of the paper.
Case $P = \mathbf{skip}$. This case is trivial.
Case $P = \mathbf{tell}(c)$. If $\langle \vec{x}; \mathbf{tell}(c); d\rangle \longrightarrow \langle \vec{x}; \mathbf{skip}, d\rangle$ then it must be the case that $d \cong d \sqcup c$ and $d \vdash c$. We conclude $\exists\vec{x}(d).[\![\mathbf{skip}]\!] \subseteq \exists\,\vec{x}([\![\mathbf{tell}(c)]\!])$.

Case $P = (\mathbf{local}\,\vec{x}; c)\,Q$. Consider the following derivation

$$\langle \vec{y}; (\mathbf{local}\,\vec{x})\,Q; d\rangle \longrightarrow \langle \vec{y} \cup \vec{x}; Q; d\rangle \longrightarrow^* \langle \vec{y} \cup \vec{x}'; Q'; d'\rangle \not\longrightarrow$$

where, by alpha-conversion, $\vec{x} \cap \vec{y} = \emptyset$ and $\vec{x} \cap fv(d) = \emptyset$. Assume that $\exists \vec{y}(d) \cong \exists \vec{y} \exists \vec{x}'(d')$. Since the derivation starting from Q is shorter than that starting from P, we conclude $\exists \vec{y}(d).\exists\, \vec{y}, \vec{x}' [\![F(Q')]\!] \subseteq \exists\, \vec{x}, \vec{y} [\![Q]\!]$.
Case $P = \mathbf{next}\,Q$. This case is trivial since $d.[\![Q]\!] \subseteq [\![P]\!]$ for any d.
Case $P = \mathbf{unless}\ c\ \mathbf{next}\,Q$. We distinguish two cases: (1) If $d \vdash c$, then we have $\langle \vec{x}; \mathbf{unless}\ c\ \mathbf{next}\,Q; d\rangle \longrightarrow \langle \vec{x}; \mathbf{skip}; d\rangle \not\longrightarrow$ and we conclude $\exists\, \vec{x}(d).[\![\mathbf{skip}]\!] \subseteq \exists\, \vec{x} [\![\mathbf{unless}\ c\ \mathbf{next}\ P]\!]$. (2), the case when $d \nvdash c$ is similar to the case of $P = \mathbf{next}\,Q$.
□

Lemma 5 (*Completeness*)
Let $\mathcal{D}.P$ be a locally independent program s.t. $d.s \in [\![P]\!]$. If $P \xrightarrow{(d,d')} R$ then $d' \cong d$ and $s \in [\![R]\!]$.

Proof
Assume that P is locally independent, $d.s \in [\![P]\!]$ and there is a derivation of the form $\langle \vec{x}; P; d\rangle \longrightarrow^* \langle \vec{x}'; P'; d'\rangle \not\longrightarrow$. We shall prove that $\exists x(d) \cong \exists \vec{x}'(d')$ and $s \in \exists\, \vec{x}' [\![F(P')]\!]$. We proceed by induction on the lexicographical order on the length of the internal derivation ($\longrightarrow^*$) and the structure of P, where the predominant component is the length of the derivation. The locally independent condition is used for the case $P = (\mathbf{local}\,\vec{x}; c)\,Q$. We present here the missing cases in the body of the paper.
Case skip. This case is trivial
Case $P = \mathbf{tell}(c)$. This case is trivial since it must be the case that $d \vdash c$ and hence $d \sqcup c \cong d$.
Case $P = \mathbf{next}\,Q$. This case is trivial since $\langle \vec{x}; P; d\rangle \not\longrightarrow$ for any d and $\vec{x}$ and $F(P) = Q$.
Case $P = \mathbf{unless}\ c\ \mathbf{next}\,Q$.If $d \vdash c$ the case is trivial. If $d \nvdash c$ the case is similar to that of $P = \mathbf{next}\,Q$.
Case $P = p(\vec{t})$. Assume that $p(\vec{x}) : -Q \in \mathcal{D}$. If $d.s \in [\![p(\vec{t})]\!]$ then $d.s \in [\![Q[\vec{t}/\vec{x}]]\!]$. By using the rule $\mathrm{R}_{\mathrm{CALL}}$ we can show that there is a derivation

$$\langle \vec{y}; p(\vec{x}); d\rangle \longrightarrow \langle \vec{y}; Q[\vec{t}/\vec{x}]; d\rangle \longrightarrow^* \langle \vec{y}'; Q'; d'\rangle \not\longrightarrow$$

By inductive hypothesis we know that $\exists y'(d') \cong \exists \vec{y}(d)$ and $s \in \exists\, \vec{y}' [\![F(Q')]\!]$.
□

Appendix C Detailed Proofs Section 4

Theorem 5 (*Soundness of the approximation*)
Let $(\mathcal{C}, \alpha, \mathcal{A})$ be a description and $\mathbf{A}$ be upper correct w.r.t. $\mathbf{C}$. Given a `utcc` program $\mathcal{D}.P$, if $s \in [\![P]\!]$ then $\alpha(s) \in [\![P]\!]^\alpha$.

Proof

Let $d_\alpha.s_\alpha = \alpha(d.s)$ and assume that $d.s \in [\![P]\!]$. Then, $d.s \in [\![P]\!]_I$ where I is the lfp of $T_{\mathcal{D}}$. By the continuity of $T_{\mathcal{D}}$, there exists n s.t. $I = T^n_{\mathcal{D}}(I_\perp)$ (the n-th application of $T_{\mathcal{D}}$). We proceed by induction on the lexicographical order on the pair n and the structure of P, where the predominant component is the length n. We present here the missing cases in the body of the paper.
Case $P = \mathbf{skip}$. This case is trivial.
Case $P = \mathbf{tell}(c)$. We must have $d \vdash c$ and by monotonicity of α, $d_\alpha \vdash^\alpha \alpha(c)$. We conclude $d_\alpha.s_\alpha \in [\![\mathbf{tell}(c)]\!]^\alpha$.
Case $P = Q \parallel R$. We must have that $s \in [\![Q]\!]$ and $s \in [\![R]\!]$. By inductive hypothesis we know that $s_\alpha \in [\![Q]\!]^\alpha$ and $s_\alpha \in [\![R]\!]^\alpha$ and then, $s_\alpha \in [\![Q \parallel R]\!]^\alpha$.
Case $P = (\mathbf{local}\,\vec{x})\,Q$. It must be the case that there exists $d'.s'$ $\vec{x}$-variant of $d.s$ s.t. $d'.s' \in [\![Q]\!]$. Then, by (structural) inductive hypothesis $\alpha(d'.s') \in [\![Q]\!]^\alpha$. We conclude by using the properties of α in Definition 12 to show that $\exists^\alpha \vec{x}(\alpha(d.s)) = \exists^\alpha \vec{x}(\alpha(d'.s'))$, i.e., $\alpha(d.s)$ and $\alpha(d'.s')$ are $\vec{x}$-variants, and then, $d_\alpha.s_\alpha \in [\![(\mathbf{local}\,\vec{x})\,Q]\!]^\alpha$.
Case $P = \mathbf{next}\,Q$. We know that $s \in [\![Q]\!]$ and by inductive hypothesis $\alpha(s) \in [\![Q]\!]^\alpha$. We then conclude $d_\alpha.s_\alpha \in [\![P]\!]^\alpha$.
Case $P = \mathbf{unless}\ c\ \mathbf{next}\,Q$. This case is trivial since $\mathcal{A}$ approximates every possible concrete computation. □

Appendix D Auxiliary results

Proposition 5
Let P be a process such that $\vec{x} \cap fv(P) = \emptyset$ and let $d.s \in [\![P]\!]$. If $d'.s'$ is an $\vec{x}$-variant of $d.s$ then $d'.s' \in [\![P]\!]$.

Proof
The proof proceeds by induction on the structure of P. We shall use the notation $c(\vec{y})$ and $P(\vec{y})$ to denote constraints and processes where the free variables are exactly $\vec{y}$ and we shall assume that $\vec{y} \cap \vec{x} = \emptyset$. We assume that $d.s \in [\![P(\vec{y})]\!]$ and $d'.s'$ is an $\vec{x}$-variant of $d.s$. We consider the following cases. The others are easy.
Case $P = \mathbf{when}\ c(\vec{y})\ \mathbf{do}\ Q(\vec{y})$. If $d' \vdash c(\vec{y})$ then, by monotonicity, $\exists\vec{x}(d') \vdash \exists\vec{x}(c(\vec{y}))$ and then $\exists\vec{x}(d) \vdash c(\vec{y})$. Hence, it must be the case that $d \vdash c(\vec{y})$ and $d.s \in [\![Q(\vec{y})]\!]$. By induction we conclude $d'.s' \in [\![Q(\vec{y})]\!]$. If $d' \nvdash c(\vec{y})$, then $\exists\vec{x}(d') \nvdash c(\vec{y})$ (since $\exists\vec{x}(d') \leq d'$). Hence, $d \nvdash c(\vec{y})$ and trivially, $d.s \in [\![P]\!]$ and so $d'.s' \in [\![P]\!]$.
Case $P = (\mathbf{abs}\ \vec{z}; c(\vec{z},\vec{y}))\,Q(\vec{z},\vec{y})$. We know that $d.s \in \forall\,\vec{z}[\![\mathbf{when}\ c(\vec{z},\vec{y})\ \mathbf{do}\ Q(\vec{z},\vec{y})]\!]$. By definition of the operator $\forall(\cdot)$, $\exists\vec{x}(d.s) \in [\![P]\!]$. Since $\exists\vec{x}(d'.s') \cong \exists\vec{x}(d.s)$ we conclude $d'.s' \in [\![P]\!]$. □

Proposition 6
If $\vec{x} \cap fv(P) = \emptyset$ then $[\![P]\!] = \exists\,\vec{x}[\![P]\!]$.

Proof
The case $[\![P]\!] \subseteq \exists\,\vec{x}[\![P]\!]$ is trivial by the definition of $\exists(\cdot)$. The case $\exists\,\vec{x}[\![P]\!] \subseteq [\![P]\!]$, follows directly from Proposition 5. □

Proposition 7
If $\vec{x} \notin fv(Q)$ then $\exists\, \vec{x}([\![P]\!] \cap [\![Q]\!]) = \exists\, \vec{x}([\![P]\!]) \cap [\![Q]\!]$.

Proof
($\subseteq$): Let $d.s \in \exists\, \vec{x}([\![P]\!] \cap [\![Q]\!])$. Then, there exists an $\vec{x}$-variant $d'.s'$ s.t. $d'.s' \in [\![P]\!] \cap [\![Q]\!]$. Then, $d.s \in \exists\, \vec{x}([\![P]\!])$ (by definition) and $d.s \in [\![Q]\!]$ by Proposition 5.
($\supseteq$): Let $d.s \in \exists\, \vec{x}([\![P]\!]) \cap [\![Q]\!]$. Then, there exists $d'.s'$ $\vec{x}$-variant of $d.s$ s.t. $d'.s' \in [\![P]\!]$. By Proposition 5, $d'.s' \in [\![Q]\!]$ and therefore, $d.s \in \exists\, \vec{x}([\![P]\!] \cap [\![Q]\!])$. □

In Theorem 5, the proof of the **abs** case requires the following auxiliary results (similar to those in the concrete semantics).

Observation 5 (*Equality and $\vec{x}$-variants*)
Let s_α and w_α be $\vec{x}$-variants such that $(d^\alpha_{\vec{x}\vec{t}})^\omega \leq^\alpha s_\alpha$, $(d^\alpha_{\vec{x}\vec{t}})^\omega \leq^\alpha w_\alpha$ and $adm(\vec{x}, \vec{t})$. Then $s_\alpha \cong^\alpha w_\alpha$.

Proof
Let $c_\alpha = s_\alpha(i)$ and $d_\alpha = w_\alpha(i)$ with $i \geq 1$. We shall prove that $c_\alpha \vdash^\alpha d_\alpha$ and $d_\alpha \vdash^\alpha c_\alpha$. We know that $c_\alpha \sqcup^\alpha d^\alpha_{\vec{x}\vec{t}} \cong^\alpha c_a$ and $d_\alpha \sqcup^\alpha d^\alpha_{\vec{x}\vec{t}} \cong^\alpha d_\alpha$. We also know that $\exists^\alpha \vec{x}(c_\alpha \sqcup^\alpha d^\alpha_{\vec{x}\vec{t}}) \cong^\alpha \exists^\alpha \vec{x}(d_\alpha \sqcup^\alpha d^\alpha_{\vec{x}\vec{t}})$. Since $c_\alpha \vdash^\alpha \exists^\alpha \vec{x}(c_\alpha)$, we can show that $c_\alpha \vdash^\alpha \exists^\alpha \vec{x}(d_\alpha \sqcup^\alpha d^\alpha_{\vec{x}\vec{t}})$. Furthermore, $\exists^\alpha \vec{x}(d_\alpha \sqcup^\alpha d^\alpha_{\vec{x}\vec{t}}) \sqcup^\alpha d^\alpha_{\vec{x}\vec{t}} \vdash^\alpha d_\alpha$ (see Notation 2). Hence, we conclude $c_\alpha \vdash^\alpha d_\alpha$. The proof of $d_\alpha \vdash^\alpha c_\alpha$ is analogous. □

Proposition 8
$s_\alpha \in \forall\, \vec{x}([\![P]\!]^\alpha_X)$ if and only if $s \in [\![P[\vec{t}/\vec{x}]]\!]^\alpha_X$ for all admissible substitution $[\vec{t}/\vec{x}]$.

Proof
($\Rightarrow$)Let $s_\alpha \in \forall\, \vec{x}([\![P]\!]^\alpha_X)$ and s'_α be an $\vec{x}$-variant of s_α s.t. $(d^\alpha_{\vec{x}\vec{t}})^\omega \leq^\alpha s'_\alpha$ where $adm(\vec{x}, \vec{t})$. By definition of $\forall$, we know that $s'_\alpha \in [\![P]\!]^\alpha_X$. Since $(d^\alpha_{\vec{x}\vec{t}})^\omega \leq^\alpha s'_\alpha$ then $s'_\alpha \in [\![P]\!]^\alpha_X \cap \uparrow((d^\alpha_{\vec{x}\vec{t}})^\omega)$. Hence, $s_\alpha \in \exists^\alpha\, \vec{x}([\![P]\!]^\alpha_X \cap \uparrow((d^\alpha_{\vec{x}\vec{t}})^\omega))$ and we conclude $s_\alpha \in [\![P[\vec{t}/\vec{x}]]\!]^\alpha_X$.
($\Leftarrow$) Let $[\vec{t}/\vec{x}]$ be an admissible substitution. Suppose, to obtain a contradiction, that $s_\alpha \in [\![P[\vec{t}/\vec{x}]]\!]^\alpha_X$, there exists s'_α $\vec{x}$-variant of s_α s.t. $(d^\alpha_{\vec{x}\vec{t}})^\omega \leq^\alpha s'_\alpha$ and $s'_\alpha \notin [\![P]\!]^\alpha_X$ (i.e., $s_\alpha \notin \forall\, \vec{x}([\![P]\!]^\alpha_X)$). Since $s_\alpha \in [\![P[\vec{t}/\vec{x}]]\!]^\alpha_X$ then $s_\alpha \in \exists^\alpha\, \vec{x}([\![P]\!]^\alpha_X \cap \uparrow(d^\alpha_{\vec{x}\vec{t}})^\omega)$. Therefore, there exists s''_α $\vec{x}$-variant of s_α s.t. $s''_\alpha \in [\![P]\!]^\alpha_X$ and $d^{\alpha\,\omega}_{\vec{x}\vec{t}} \leq^\alpha s''_\alpha$. By Observation 5, $s'_\alpha \cong^\alpha s''_\alpha$ and thus, $s'_\alpha \in [\![P]\!]^\alpha_X$, a contradiction. □