Abstract interpretation of temporal concurrent constraint programs

MORENO FALASCHI
Dipartimento di Ingegneria dell’Informazione e Scienze Matematiche
Università di Siena, 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, Palaiseau, 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 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 that 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 also make 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 a large body of reasoning techniques of both process calculi and logic. In fact, ccp-based calculi have been used successfully in the modeling and verification of several concurrent scenarios such as biological, security, timed, reactive, and stochastic systems (Saraswat et al. 1991; Saraswat et al. 1994; Nielsen et al. 2002a; Jagadeesan et al. 2005; Olarte and Valencia 2008b) (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 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 \((\text{abs } \vec{x}; c) P\) can be neatly characterized as closure operators. This characterization is shown to be somehow dual to the semantics for the local operator \((\text{local } \vec{x}) P\) 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 that 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 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, 1997b); Nielsen et al. (2002a); and Olarte and Valencia (2008a) to give semantic characterization of ccc calculi and from the works in Falaschi et al. (1993); Codish et al. (1994); Falaschi et al. (1997a); Zaffanella et al.
(1997); and 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 τcc programs.

The rest of the paper is organized as follows. Section 2 recalls the notion of constraint system and the operational semantics of τcc 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 τcc and utcc programs. The three instances and the applications of the framework are presented in Section 5. Section 6 concludes the paper.

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 \( C = \langle C, \leq, \sqcup, t, f, Var, \exists, D \rangle \) s.t.

- \( \langle C, \leq, \sqcup, t, f \rangle \) is a lattice with \( \sqcup \) the lub operation (representing the logical and), and \( t, f \) the least and the greatest elements in \( C \) respectively (representing true and false).
- Elements in \( C \) are called constraints with typical elements \( c, c', d, d', \ldots \). If \( c \leq d \) and \( d \leq c \), we write \( c \cong d \). If \( c \leq d \) and \( c \not\cong d \), we write \( c < d \).
- \( Var \) is a denumerable set of variables and for each \( x \in Var \) the function \( \exists x : C \to C \) 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) \( \exists x(c \cup \exists x(d)) \cong \exists x(c) \sqcup \exists x(d) \). (4) \( \exists x\exists y(c) \cong \exists y\exists x(c) \). (5) For an increasing chain \( c_1 \leq c_2 \leq c_3, \ldots, \exists x \bigcup_{i=1}^{j} c_i \cong \bigcup_{i=1}^{j} \exists x(c_i) \).
- For each \( x, y \in Var \), the constraint \( d_{xy} \in D \) is a diagonal element and it satisfies: (1) \( d_{xy} \cong t \). (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 these in Notation 2.

Let us give some examples of constraint systems. The finite domain (FD) constraint system (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 $x = g(a) \lor y = z$. $\mathcal{H}$ 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 $gt(x, n)$ (i.e., $x > n$) and $\{gt(x, n)\}_n$ be the ascending chain $gt(x, 0) < gt(x, 1) < \ldots$. We note that $\exists x(gt(x, n)) = t$ for any $n$ and then $\bigcup \{\exists x(gt(x, n))\}_n = t$. Property (5) in Definition 1 dictates that $\exists x \bigcup \{gt(x, n)\}_n$ must be equal to $t$ (i.e., there exists an $x$ which is greater than any $n$). For that, we need a constraint, e.g., $\inf(x)$ (a non-compact element), to be the limit $\bigcup \{gt(x, n)\}_n$. We know that $\inf(x) \vdash gt(x, n)$ for any $n$ and then $\bigcup \{gt(x, n)\}_n = \inf(x)$ and $\exists x(\inf(x)) = t$ 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 $\emptyset$) to be the limit of the chain $\{gt(x, n)\}_n$.

Now consider the Kahn constraint system underlying data-flow languages where equality is assumed along with the constant $nil$ (the empty list), the predicate $\text{nempty}(x)$ ($x$ is not $nil$), and the functions $\text{first}(x)$ (the first element of $x$), $\text{rest}(x)$ ($x$ without its first element), and $\text{cons}(x, y)$ (the concatenation of $x$ and $y$). If we consider the Kahn constraint system in Saraswat et al. (1991), the constraint $c$ defined as $\{\text{first} (\text{tail}^n(x)) = \text{first} (\text{tail}^n(y)) \mid n \geq 0\}$ 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 no 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 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 $c^\omega$ to denote the sequence $c.c.c....$ The length of $s$ is denoted by $|s|$ and the empty sequence by $\epsilon$. The $i$th element in $s$ is denoted by $s(i)$. We write $s \preceq s'$ iff $|s| \leq |s'|$ and for all $i \in \{1,\ldots,|s|\}$, $s'(i) \vdash s(i)$. If $|s| = |s'|$ and for all $i \in \{1,\ldots,|s|\}$ 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 $\exists x_1\exists x_2\ldots \exists x_n(c)$ and with $\exists \vec{x}(s)$ we mean the point-wise 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, we consider the following.

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 $\bigcup_{1 \leq i \leq n} x_i = t_i$. If $|\vec{x}| = |\vec{t}| = 0$, then $d_{\vec{x}\vec{t}} = \top$. 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 $E \not\vDash d_{\vec{x}\vec{t}}$.

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 $\exists \vec{x}(c \cup d_{\vec{x}\vec{t}})$ 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 $c[\vec{t}/\vec{x}] \cup d_{\vec{x}\vec{t}} \vdash c$.

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 (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 also determines 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 above example may change its value when passing from one time-unit to the next one.

**Definition 2 (tcc Processes)**

The set $\text{Proc}$ of tcc processes is built from the syntax

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

where $c$ is a compact element of the underlying constraint system. Let $\emptyset$ be a set of process declarations of the form $p(\bar{x}) := P$. A tcc program takes the form $\emptyset. P$. We assume $\emptyset$ 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 other processes. The process when $c$ do $P$ asks if $c$ can be deduced from the store. If so, it behaves as $P$. In the 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 (local $\bar{x}$) $P$ binds $\bar{x}$ in $P$ by declaring it private to $P$. It behaves like $P$, except that all the information on the variables $\bar{x}$ produced by $P$ can only be seen by $P$ and the information on the global variables in $\bar{x}$ produced by other processes cannot be seen by $P$.

The process next $P$ is a unit-delay that executes $P$ in the next time-unit. The time-out unless $c$ 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 next$^n P$ as a shorthand for next...next $P$, with next repeated $n$ times.

We extend the definition of free variables to processes as follows: $fv(\text{skip}) = \emptyset$; $fv(\text{tell}(c)) = fv(c); \ f(v(\text{when}\ c\ \text{do}\ Q) = fv(c) \cup fv(Q); \ f(v(\text{unless}\ c\ \text{next}\ Q) = fv(c) \cup fv(Q); \ f(v(Q \parallel Q')) = fv(Q) \cup fv(Q'); \ f(v(\text{local}\ \bar{x})\ Q) = fv(Q) \setminus \bar{x}; \ f(v(\text{next}\ Q)) = fv(Q); \ f(v(p(\bar{t}))) = vars(\bar{t}),$ where $vars(\bar{t})$ is the set of variables occurring in $\bar{t}$. 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 $p(\bar{x}) := P$ where $fv(P) \subseteq \bar{x}$. The call $p(\bar{t})$ reduces to $P[\bar{t}/\bar{x}]$. 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 \( \text{bang}_P() := P \parallel \text{next}\text{bang}_P() \). Hence, \( !P \) means \( P \parallel \text{next}P \parallel \text{next}^2P \ldots \).

### 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, for example, in the \( \pi \)-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 \( \text{out}(\cdot) \) be a constraint and let \( P = \text{when} \text{out}(x) \text{ do } R \) be a system that must react when receiving a stimulus (i.e., an input) of the form \( \text{out}(n) \) for \( n > 0 \). We note that \( P \) in a store \( \text{out}(42) \) does not execute \( R \) since \( \text{out}(42) \not\models \text{out}(x) \).

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 \( \text{out}(t) \) is entailed by the store.

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

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

From a programming point of view, we can see the variables \( \vec{x} \) in the abstraction \( (\text{abs } \vec{x}; c)P \) 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 also adhere 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

$$ P = (\text{local } x) (\text{tell}(\text{out}(x)) \parallel P') \quad Q = (\text{abs } z; \text{out}(z)) Q' $$

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 Structural operational semantics (SOS)

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

**Definition 4 (Structural congruence)**

Let $≡$ be the smallest congruence satisfying:

1. $P ≡ Q$ if they differ only by a renaming of bound variables (alpha-conversion);
2. $P \parallel \text{skip} ≡ P$;
3. $P \parallel Q ≡ Q \parallel P$; and
4. $P \parallel (Q \parallel R) ≡ (P \parallel Q) \parallel R$.

The congruence relation $≡$ is extended to configurations by decreeing that $⟨\vec{x}; P; c⟩ ≡ ⟨\vec{y}; P; d⟩$ iff $(\text{local } \vec{x}) P ≡ (\text{local } \vec{y}) Q$ and $∃\vec{x}(c) ≡ ∃\vec{y}(d)$.

Transitions are given by the relations $→$ and $→*$ in Figure 1. The *internal* transition $⟨\vec{x}; P; c⟩ → ⟨\vec{x}'; P'; c'⟩$ should be read as “$P$ with store $c$ reduces, in one internal step, to $P'$ with store $c'$.” We shall use $→*$ as the reflexive and transitive closure of $→$. If $γ → γ'$ and $γ' ≡ γ''$, we write $γ →≡ γ''$. Similarly for $→*$.

The *observable transition* $P \overset{(c,d)}{→} R$ 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 $R_{\text{OBS}}$. Before explaining such rules, let us introduce the following notation needed for $R_{\text{STRVAR}}$.

**Notation 4 (Normal form)**

We observe that the store $c$ in a configuration takes the form $∃\vec{x}_1(d_1) \sqcup ... \sqcup ∃\vec{x}_n(d_n)$ where each $\vec{x}_i$ may be an empty set of variables. The normal form of $c$, notation
Abstract interpretation of temporal CCP

Fig. 1. SOS. In $R_{STR}$, $\equiv$ is given in Definition 4. In $R_{ABS}$ and $R_{CALL}$, $\text{adm}(\vec{x}, \vec{t})$ is defined in Notation 2. In $R_{ABS}$, $E$ is assumed to be a set of diagonal elements and $\nabla d\vec{y}$ is defined in Convention 1. In $R_{STRVAR}$, $\text{nf}(c)$ is defined in Notation 4.

$\text{nf}(c)$ 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 $c \equiv \text{nf}(c)$.

- $R_{TELL}$ says that the process $\text{tell}(c)$ adds $c$ to the current store $d$ (via the lub operator of the constraint system) and then evolves into $\text{skip}$.
- $R_{PAR}$ says that if $P$ may evolve into $P'$, this reduction also takes place when running in parallel with $Q$.
- The process $(\text{local} \, \vec{y}) \, Q$ adds $\vec{y}$ to the local variables of the configuration and then evolves into $Q$. The side conditions of the rule $R_{LOC}$ 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 $(\text{abs} \, \vec{y}; c; E) \, Q$ instead of $(\text{abs} \, \vec{y}; c; \emptyset) \, Q$. If $d$ entails $c[\vec{t}/\vec{y}]$, then $P[\vec{t}/\vec{y}]$ is executed (Rule $R_{ABS}$). Moreover, the abstraction persists in the current time interval to allow other potential replacements of $\vec{y}$ in $P$. Note that $E$ is augmented with $d\vec{y}$ and the side condition $E \nabla d\vec{y}$ 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., $(\text{local} \, \vec{y}) \, (\text{tell}(d\vec{y}) \parallel P)$.
- Rule $R_{STRVAR}$ allows us to open the scope of existentially quantified constraints in the store (see Example 3 below). If $\gamma$ reduces to $\gamma'$ using this rule, then $\gamma \equiv \gamma'$.
- Rule $R_{STR}$ 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
\[\equiv\] on constraints. We shall discuss the reasons behind this choice in Example 3.
- What we observe from \(p(i)\) is \(P[p/\bar{x}]\) where the formal parameters are substituted by the actual parameter (Rule R_CALL).
- Since the process \(P = \text{unless } c\ \text{next} \ Q\) executes \(Q\) in the next time-unit only if the final store at the current time-unit does not entail \(c\), in the rule R_UNL \(P\) evolves into skip if the current store \(d\) entails \(c\).

The observable transition relation, rule ROBS says that an observable transition from \(P\) labeled with \((c, \exists \bar{x}(d))\) is obtained from a terminating sequence of internal transitions from \((\emptyset; P; c)\) to \((\bar{x}; Q; d)\). The process to be executed in the next time interval is \((\text{local } \bar{x})\ F(Q)\) (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. Note that the output of a process hides the local variables \((\exists \bar{x}(d))\) and those variables are also hidden in the next time-unit \((\text{local } \bar{x})\ F(Q)\).

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 \(\exists w(\text{out}(w))\):

1. \((\emptyset; P \parallel Q; \exists w(\text{out}(w))) \rightarrow^* (\{x\}; \text{tell}(\text{out}(x)) \parallel P' \parallel Q; \exists w(\text{out}(w)))\)
2. \(\rightarrow^* (\{x\}; P' \parallel Q; \exists w(\text{out}(w)) \sqcup \text{out}(x))\)
3. \(\rightarrow^* (\{x, w\}; P' \parallel Q; \exists w(\text{out}(w)) \sqcup \text{out}(x))\)
4. \(\rightarrow^* (\{x, w\}; P' \parallel Q_1 \parallel Q'[w/z]; \exists w(\text{out}(w)) \sqcup \text{out}(x))\)
5. \(\rightarrow^* (\{x, w\}; P' \parallel Q_2 \parallel Q'[w/z]; \exists w(\text{out}(w)) \sqcup \text{out}(x))\)

where \(Q_1 = (\text{abs } z; \text{out}(z); \{d_{wz}\}) \ Q'\) and \(Q_2 = (\text{abs } z; \text{out}(z); \{d_{wz}, d_{cz}\}) \ Q'\). 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 R.StringVar to open the scope of \(w\) in the store \(\exists w(\text{out}(w))\). Let \(c_1 = \exists w(\text{out}(w)) \sqcup \text{out}(x)\) (store in line 2) and \(c_2 = \text{out}(x)\). We know that \(c_1 \equiv c_2\). As we said before, Rule R_Str allows us to replace structural congruent processes (\(\equiv\)), but it does not modify the store via the relation \(\equiv\) on constraints. The reason is that if we replace \(c_1\) with \(c_2\) in line 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 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 the 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.

Next lemma states some fundamental properties of the internal relation. The proof follows from simple induction on the inference \(\gamma \rightarrow \gamma'\).
Lemma 1 (Properties of \( \rightarrow \))
Assume that \( \langle \tilde{x}; P ; c \rangle \rightarrow \langle \tilde{x}'; Q ; d \rangle \). Then \( \tilde{x} \subseteq \tilde{x}' \). Furthermore:

1. (Internal extensiveness): \( \exists \tilde{x}'(d) \vdash \exists \tilde{x}(c) \), i.e., the store can only be augmented.
2. (Internal potentiality): If \( e \vdash c \) and \( d \vdash e \), then \( \langle \tilde{x}; P ; e \rangle \rightarrow= \langle \tilde{x}'; Q ; d \rangle \), i.e., a stronger store triggers more internal transitions.
3. (Internal restartability): \( \langle \tilde{x}; P ; d \rangle \rightarrow= \langle \tilde{x}'; Q ; d \rangle \).

2.4.1 Input–output behavior

Recall that \( \text{tcc} \) and \( \text{utcc} \) allow 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 \( \text{tcc} \), some \( \text{utcc} \) processes may exhibit infinitely many internal reductions during a time-unit due to the \( \text{abs} \) operator.

Example 4 (Infinite behavior)
Consider a constant symbol “\( a \)” a function symbol \( f \), and a unary predicate (constraint) \( c(\cdot) \), and let \( Q = (\text{abs } x ; c(x)) \text{tell}(c(f(x))) \). 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. \( P \xrightarrow{(w,w')} \)). We define the input–output behavior of \( P \) as \( \text{io}^{\text{fin}}(P) = \text{io}^{\text{fin}}(P) \cup \text{io}^{\text{inf}}(P) \), where

\[
\text{io}^{\text{fin}}(P) = \{ (s,s') | P \xrightarrow{(s,s')} \} \text{ for } s, s' \in \mathcal{E}^* \\
\text{io}^{\text{inf}}(P) = \{ (w,w') | P \xrightarrow{(w,w')} \} \text{ for } w, w' \in \mathcal{E}^o 
\]

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

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) show 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 \((s, w), (s, w') \in \text{io}(P)\), 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 \( P = \\text{unless } c \text{ next } Q \) and \( s \preceq s' \). If \((s, w), (s', w') \in \text{io}(P)\), it may be the case that \( w \npreceq w' \). For example, take \( Q = \text{tell}(d) \), \( s = t^\omega \) and \( s' = c \cdot t^\omega \). The reader can verify that \( w = t \cdot d \cdot t^\omega, w' = c \cdot t^\omega \) and then \( w \npreceq w' \).

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

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

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 \(T_D\), which computes in a bottom-up fashion the interpretation of each process definition \(p(\vec{x}) : P\) in \(D\). Such an interpretation is given in terms of the set of the quiescent sequences for \(p(\vec{x})\).

Assume a utcc program \(D.P\). We shall denote the set of process names with their formal parameters in \(D\) as \(ProcHeads\). We shall call Interpretations the set of functions in the domain \(ProcHeads \rightarrow \mathcal{P}(\mathcal{C}_\omega)\). We shall define the semantics as a function \(\llbracket \cdot \rrbracket : (ProcHeads \rightarrow \mathcal{P}(\mathcal{C}_\omega)) \rightarrow (Proc \rightarrow \mathcal{P}(\mathcal{C}_\omega))\), which, given an interpretation \(I\), associates to each process a set of sequences of constraints.

Before defining the semantics, we introduce the following notation.

**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 $S \subseteq \mathcal{G}^\omega$, we define:

\[
\begin{align*}
\exists \hat{x}(S) &= \{ s \in \mathcal{G}^\omega \mid \text{there exists } s' \in S \text{ s.t. } \exists \hat{x}(s) \equiv \exists \hat{x}(s') \} \\
\forall \hat{x}(S) &= \{ \exists \hat{y}(s) \in S \mid \hat{y} \in \text{Var}, s \in S \text{ and for all } s' \in \mathcal{G}^\omega, \text{ if } \exists \hat{x}(s) \equiv \exists \hat{x}(s'), \text{ then } s' \in S \}
\end{align*}
\]

The operators above are used to define the semantic equations in Figure 2 and explained in the following. Recall that $\llbracket P \rrbracket_I$ aims at capturing the strongest postcondition (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 $\mathcal{G}^\omega$ (equation D_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 D_TELL). If neither $P$ nor $Q$ can add any information to $s$, then $s$ is quiescent for $P \parallel Q$ (equation D_PAR).

We say that $s$ is an $\hat{x}$-variant of $s'$ if $\exists \hat{x}(s) \equiv \exists \hat{x}(s')$, i.e., $s$ and $s'$ differ only on the information of $\hat{x}$. Let $S = \exists \hat{x}(S')$. We note that $s \in S$ if there is an $\hat{x}$-variant $s'$ of $s$ in $S'$. Therefore, a sequence $s$ is quiescent for $Q = (\text{local } \hat{x})P$ if there exists an $\hat{x}$-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 D_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 $c \in \mathcal{G}$ (equation D_NEXT). Recall that the process $Q = \text{unless } c \text{ next } P$ 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 D_UNL). This equation can be equivalently written as $\mathcal{G} \llbracket P \rrbracket_I \cup \uparrow c : \mathcal{G}^\omega$.

Recall that the interpretation $I$ maps process names to sequences of constraints. Then the meaning of $p(i)$ is directly given by the interpretation $I$ (Rule D_CALL).

Let $Q = \text{when } c \text{ do } P$. 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 D_ASK). In some cases, for the sake of presentation, we may write this equation as follows:

\[
\llbracket \text{when } c \text{ do } P \rrbracket_I = \{ d.s \mid \text{if } d \vdash c \text{ then } d.s \in \llbracket P \rrbracket_I \}
\]

Before explaining the Rule D_ABS, let us show some properties of $\forall \hat{x}(\cdot)$. First, we note that the $\hat{x}$-variables satisfying the condition $d_{\hat{x}t}^o \leq s$ in the definition of $\forall$ are equivalent (see the proof in Appendix B.).
Abstract interpretation of temporal CCP

327

Observation 1 (Equality and \(\bar{x}\)-variants)

Let \(S \subseteq \mathcal{C}^\omega\), \(\bar{z} \subseteq \text{Var}\), and \(s, w \in \mathcal{C}^\omega\) be \(\bar{x}\)-variants such that \(d_{\bar{x}\bar{t}}^s \leq s\), \(d_{\bar{x}\bar{t}}^w \leq w\) and \(\text{adm}(\bar{x}, \bar{t})\). (1) \(s \equiv w\). (2) \(\exists \exists (s) \in \mathcal{V} \ \bar{x}(S)\) if \(s \in \mathcal{V} \ \bar{x}(S)\).

Now we establish the correspondence between the sets \(\mathcal{V} \ \bar{x}(\llbracket P \rrbracket_I)\) and \(\llbracket P [\bar{t}/\bar{x}] \rrbracket_I\) which is fundamental to understand the way we defined the operator \(\mathcal{V}\).

Proposition 1

\(s \in \mathcal{V} \ \bar{x}(\llbracket P \rrbracket_I)\) if and only if \(s \in \llbracket P [\bar{t}/\bar{x}] \rrbracket_I\) for all admissible substitution \([\bar{t}/\bar{x}]\).

Proof

\((\Rightarrow)\) Let \(s \in \mathcal{V} \ \bar{x}(\llbracket P \rrbracket_I)\) and \(s'\) be an \(\bar{x}\)-variant of \(s\) s.t. \(d_{\bar{x}\bar{t}}^s \leq s'\) where \(\text{adm}(\bar{x}, \bar{t})\). By definition of \(\mathcal{V}\), we know that \(s' \in \llbracket P \rrbracket_I\). Since \(d_{\bar{x}\bar{t}}^s \leq s'\), \(s' \in \llbracket P \rrbracket_I \cap \llbracket d_{\bar{x}\bar{t}} \rrbracket\). Hence, \(s \in \exists \ \bar{x}(\llbracket P \rrbracket_I) \cap \llbracket d_{\bar{x}\bar{t}} \rrbracket\), and we conclude \(s \in \llbracket P [\bar{t}/\bar{x}] \rrbracket_I\).

\((\Leftarrow)\) Let \([\bar{t}/\bar{x}]\) be an admissible substitution. Suppose, to obtain a contradiction, that \(s \in \llbracket P [\bar{t}/\bar{x}] \rrbracket_I\), there exists \(s'\) \(\bar{x}\)-variant of \(s\) s.t. \(d_{\bar{x}\bar{t}}^s \leq s'\) and \(s' \notin \llbracket P \rrbracket_I\) (i.e., \(s \notin \mathcal{V} \ \bar{x}(\llbracket P \rrbracket_I)\)). Since \(s \in \llbracket P [\bar{t}/\bar{x}] \rrbracket_I\), \(s \in \exists \ \bar{x}(\llbracket P \rrbracket_I) \cap \llbracket d_{\bar{x}\bar{t}} \rrbracket\). Therefore, there exists \(s''\) \(\bar{x}\)-variant of \(s\) s.t. \(s'' \in \llbracket P \rrbracket_I\) and \(d_{\bar{x}\bar{t}}^s \leq s''\). By Observation 1, \(s' \equiv s''\) and thus \(s' \in \llbracket P \rrbracket_I\), a contradiction. \(\square\)

A sequence \(d.s\) is quiescent for the process \(Q = (\text{abs } x; c) P\) if for all admissible substitution \([\bar{t}/\bar{x}]\), either \(d \not\vdash c[\bar{t}/\bar{x}]\) or \(d.s\) is also quiescent for \(P [\bar{t}/\bar{x}]\), i.e., \(d.s \in \mathcal{V} \ \bar{x}(\llbracket \text{when } c \text{ do } P \rrbracket)\) (rule DABS). Note that we can simply write equation DABS by unfolding the definition of DASK as follows:

\[
\llbracket (\text{abs } \bar{x}; c) P \rrbracket_I = \mathcal{V} \ \bar{x}(\llbracket \text{abs } \bar{x}; c \llbracket^\omega \cup (\llbracket c \llbracket^\omega \cap \llbracket P \rrbracket_I))
\]

The reader may wonder why the operator \(\mathcal{V}\) (resp. Rule DABS) is not entirely dual w.r.t. \(\exists\) (resp. Rule DLOC), i.e., why we only consider \(\bar{x}\)-variants entailing \(d_{\bar{x}\bar{t}}\) where \([\bar{t}/\bar{x}]\) is an admissible substitution. To explain this issue, let \(Q = (\text{abs } x; c) P\) where \(c = \text{out}(x)\) and \(P = \text{tell}(\text{out}'(x))\). We know that

\[
s = (\text{out}(a) \land \text{out}'(a)). t^\omega \in \text{sp}(Q)
\]

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

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

Let \(c' = \text{out}(a) \land \text{out}'(a) \land \text{out}(x)\) and \(s' = c'. t^\omega\). Note that \(s'\) is an \(x\)-variant of \(s\), \(s'(1) \vdash c\) but \(s' \notin \llbracket P \rrbracket_I\) (since \(c' \not\vdash \text{out}'(x)\)). Then \(s \notin \llbracket Q \rrbracket_I\) under this naive definition of \(\llbracket Q \rrbracket_I\). We thus consider only the \(\bar{x}\)-variants \(s'\) s.t. each element of \(s'\) entails \(d_{\bar{x}\bar{t}}\). Intuitively, this condition requires that \(s'(1) \vdash c \sqcup d_{\bar{x}\bar{t}}\) in equation DABS and hence that \(s'(1) \vdash c[\bar{t}/\bar{x}]\). Furthermore, \(s \in \llbracket P [\bar{t}/\bar{x}] \rrbracket_I\) realizes the operational intuition that \(P\) runs under the substitution \([\bar{t}/\bar{x}]\). The operational rule RSTRVAR makes also echo in the design of our semantics: the operator \(\mathcal{V}\) considers constraints of the form \(\exists \bar{z}(s)\), where \(\bar{z}\) 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 $P = \text{when out}(x) \text{ do tell}(\text{out}'(x))$, $Q = (\text{abs } \bar{x}; \text{out}(x)) \text{ tell}(\text{out}'(x))$. We know that $[Q]_I = \forall x([P]_I)$. Assume that $d.s \in [P]_I$. Then $d$ must be in the set:

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

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

3.1 Compositional semantics

We choose as semantic domain $\mathcal{E} = (E, \sqsubseteq^c)$, where $E = \{ X \mid X \in \mathcal{P}(\omega) \}$ and $\omega \in X$. The bottom of $\mathcal{E}$ is then $\mathcal{E}^0$ (the set of all the sequences) and the top element is the singleton $\{ \omega \}$ (recall that $\omega$ is the greatest element in $\langle \mathcal{E}, \sqsubseteq \rangle$). Given two interpretations $I_1$ and $I_2$, we write $I_1 \sqsubseteq^c I_2$ if 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:

$$T_{\mathcal{D}}(I)(p(\bar{t})) = [[Q][\bar{t}/\bar{x}]_I]_I$$

if $p(\bar{x}) : Q \in \mathcal{D}$

We shall use $[[P]]_I$ to represent $[[P]]_I|_{p(T_{\mathcal{D}})}$.

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 $\mathcal{E}$.

**Lemma 3 (Properties of $\forall$)**

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

**Proof**

The proofs of (1), (2), and (3) are straightforward from the definition of $\forall \bar{x}$. The proof of (4) proceeds as follows. Assume a nonempty 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 $\bigcap \forall \bar{x}(S_i) = \forall \bar{x}(\bigcap S_i)$. The “$\sqsubseteq$” part (i.e., $\sqsubseteq^c$) is trivial since $\forall$ is monotonic. As for the $\bigcap \forall \bar{x}(S_i) \subseteq \forall \bar{x}(\bigcap S_i)$ part, by extensiveness we know that $\forall \bar{x}(S_i) \subseteq S_i$ for all $S_i$ and then $\bigcap \forall \bar{x}(S_i) \subseteq \bigcap S_i$. Let $s \in \bigcap \forall \bar{x}(S_i)$. By definition we know that $s$ and all $\bar{x}$-variant $s'$ of $s$ satisfying $d_{\bar{x}}^{s'} \sqsubseteq s'$ for $\text{adm}(\bar{x}, \bar{i})$ belong to $\bigcap \forall \bar{x}(S_i)$ and then in $\bigcap S_i$. Hence, $s \in \forall \bar{x}(\bigcap S_i)$ and we conclude $\bigcap \forall \bar{x}(S_i) \subseteq \forall \bar{x}(\bigcap S_i)$. □
Abstract interpretation of temporal CCP 329

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

Proposition 2 (Monotonicity of $[\cdot]$ and continuity of $T_D$)

Let $P$ be a process and $I_1 \subseteq_c I_2 \subseteq_c I_3 \ldots$ be an ascending chain. Then $[P]_{I_i} \subseteq_c [P]_{I_{i+1}}$ (Monotonicity). Moreover, $[P]_{I_i} = \bigcup_{I_i} [P]_{I_i}$ (Continuity).

Proof

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

Example 6 (Computing the semantics)

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

$$
\mathcal{D} = p() :\text{ tell}(\text{out}_a(x)) \parallel \text{ next tell}(\text{out}_a(y))
q() : (\text{abs } z ; \text{out}_a(z)) (\text{tell}(\text{out}_b(z))) \parallel \text{ next } q()
$$

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 $[r(i)]$ can be found in Figure 3. Let $s \in [r(i)]$. Then it must be the case that $s \in [p()]$, and then $s(1) \vdash \text{out}_a(x)$ and $s(2) \vdash \text{out}_a(y)$. Since $r \in [q(i)]$, for $i \geq 1$, if $s(i) \vdash \text{out}_a(t)$, $s(i) \vdash \text{out}_b(t)$ for any term $t$. Hence, $s(1) \vdash \text{out}_a(x)$ and $s(2) \vdash \text{out}_b(y)$.

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 $d \equiv d'$, then $d.[R] \subseteq [P]$. 
Proof
Assume that \( \langle \hat{x}; P; d \rangle \rightarrow^* \langle \hat{x}'; P'; d' \rangle \not\rightarrow \), \( \exists \hat{x}(d) \cong \exists \hat{x}'(d') \). We shall prove that \( \exists \hat{x}(d). \exists \hat{x}'(F'(P')) \subseteq \exists \hat{x}(F(P)) \). We proceed by induction 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
\[
\langle \hat{z}; Q \parallel S, d \rangle \rightarrow^* \langle \hat{z} \cup \hat{x}_1 \cup \check{y}_1; Q_1 \parallel S_1, c_1 \cup e_1 \rangle \\
\rightarrow^* \langle \hat{z} \cup \hat{x}_2 \cup \check{y}_2; Q_1 \parallel S_1, c_1 \cup e_2 \rangle \\
\rightarrow^* \langle \hat{z} \cup \hat{x}_m \cup \check{y}_m; Q_m \parallel S_n, c_m \cup e_n \rangle \\
\not\rightarrow
\]
such that for \( i > 0 \), each \( Q_{i+1} \) (resp. \( S_{i+1} \)) is an evolution of \( Q_i \) (resp. \( S_i \); \( \hat{x}_i \) (resp. \( \check{y}_j \)) 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 \( \hat{x}_m \cap \check{y}_n = \emptyset \). We know that \( \exists \hat{z}(d) \cong \exists \hat{z}, \hat{x}_m, \check{y}_n(c_m \cup e_n) \) and from \( R_{PAR} \) we can derive:
\[
\langle \hat{z} \cup \check{y}_n; Q; d \cup e_n \rangle \rightarrow^* \langle \hat{z} \cup \hat{x}_m \cup \check{y}_n; Q_m, c_m \cup e_n \rangle \not\rightarrow \quad \text{and} \\
\langle \hat{z} \cup \hat{x}_m; S; d \cup c_m \rangle \rightarrow^* \langle \hat{z} \cup \hat{x}_m \cup \check{y}_n; S_n, c_m \cup e_n \rangle \not\rightarrow
\]
By (structural) inductive hypothesis, we know that \( \exists \hat{z}, \check{y}_n(d \cup e_n), \exists \hat{z}, \hat{x}_m, \check{y}_n(F(Q_m)) \subseteq \exists \hat{z}, \check{y}_n(F[S])) \) and also \( \exists \hat{z}, \hat{x}_m(d \cup c_m), \exists \hat{z}, \check{y}_n, \hat{x}_m(F(S_n)) \subseteq \exists \hat{z}, \check{y}_n(F[S])) \). We note that \( \exists \hat{z}(F[Q] \cap F[S]) = \exists \hat{z}(F[S]) \cap F[S] \) (see Proposition 7 in Appendix D.). Hence, from the fact that \( \hat{x}_m \cap \check{y}_n = \emptyset \), we conclude:
\[
\exists \hat{z}(d). \exists \hat{z}, \hat{x}_m, \check{y}_n(F(Q_m)) \cap F(S_n) \subseteq \exists \hat{z}(F[Q] \cap F[S])
\]

Case \( P = \text{(abs } \check{x}; c; Q) \). From the rule \( R_{ABS} \), we can show that
\[
\langle \check{y}; P; d \rangle \rightarrow^* \langle \check{y}_1; P_1 \parallel Q_1[t_i/\check{x}]; d_1 \rangle \\
\rightarrow^* \langle \check{y}_2; P_2 \parallel Q_2[t_i/\check{x} \cap \check{y}_i]; d_2 \rangle \\
\rightarrow^* \langle \check{y}_3; P_3 \parallel Q_3[t_i/\check{x} \cup \check{y}_i]; d_3 \rangle \\
\rightarrow^* \ldots \\
\rightarrow^* \langle \check{y}_n; P_n \parallel Q_n[t_i/\check{x} \cup \check{y}_i]; d_n \rangle \\
\]
where \( P_n \) takes the form \( \text{(abs } \check{x}; c; E_n) \), \( E_n = \{ d_{z_1}, ..., d_{z_n} \} \) and \( \exists \check{y}(d) \cong \exists \check{y}(d_n) \). Hence, there is a derivation (shorter than that for \( P \)) for each \( d_{z_i} \in E_n \):
\[
\langle \check{y}_i; Q_i[t_i/\check{x}]; d_i \rangle \rightarrow^* \langle \check{y}_i'; Q_i'[t_i/\check{x}]; d_i' \rangle \not\rightarrow
\]
with \( Q_i[t_i/\check{x}] = Q_i'[t_i/\check{x}] \) and \( \exists \check{y}_i(d_i) \cong \exists \check{y}_i'(d_i') \). Therefore, by inductive hypothesis,
\[
\exists \check{y}_i(d_i). \exists \check{y}_i(F(Q_i'[t_i/\check{x}]))) \subseteq \exists \check{y}(F(Q_i'[t_i/\check{x}])))
\]
for all \( d_{z_i} \in E_n \). We assume, by alpha conversion, that the variables added for each \( Q_i' \) are distinct and then their intersection is empty. Furthermore, we note that \( \exists \check{y}(d) \cong \exists \check{y}_1(d_1) \). Since \( F(P_n) = \text{skip} \), we then conclude:
\[
\exists \check{y}(d). \exists \check{y}_n(F(P_n \parallel \prod_{d_{z_i} \in E_n} Q_i'[t_i/\check{x}]))) \subseteq \exists \check{y}_1(\prod_{d_{z_i} \in E_n} Q_i'[t_i/\check{x}]))
\]
Let $d.s \in \exists \hat{y}\{ \prod_{d.s_{i}\in E_n} Q[i_{i}/\hat{x}]\}$. For an admissible $d.s_{i}$, either $d \not\vdash c[i/\hat{x}]$ or $d \vdash c[i/\hat{x}]$.

In the first case, trivially $d.s \in \\lbrack\{\text{when } c \text{ do } Q\}[i/\hat{x}]\rbrack$. In the second case, $E_n \models d.s_{i}$. Hence, $d.s \in \\lbrack Q[i/\hat{x}]\rbrack$ and $d.s \in \\lbrack\{\text{when } c \text{ do } Q\}[i/\hat{x}]\rbrack$. Here we conclude that for all admissible $[i/\hat{x}]$, $d.s \in \\lbrack\{\text{when } c \text{ do } Q\}[i/\hat{x}]\rbrack$ and by Proposition 1 we derive:

$$\exists \hat{y}(d).\exists \hat{y} F \left( \prod_{d.s_{i}\in E_n} Q_{\text{utcc}}[i_{i}/\hat{x}] \right) \subseteq \exists \hat{y} \forall \hat{x} \{\text{when } c \text{ do } Q\}$$

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

$$\langle \hat{y}; p(i); d \rangle \rightarrow \langle \hat{y}; Q[i/\hat{x}] ; d \rangle \rightarrow^* \langle \hat{y}; Q'; d' \rangle \rightarrow^*$$

where $\exists \hat{y}'(d') \equiv \exists \hat{y}(d)$. By induction $\exists \hat{y}(d).\exists \hat{y} F(Q') \subseteq \exists \hat{y} \{Q[i/\hat{x}]\}$ and we conclude $\exists \hat{y}(d).\exists \hat{y} F(Q') \subseteq \exists \hat{y} \lbrack p(i) \rbrack$. □

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. $s.s' \in \lbrack P \rbrack$.

**Proof**

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

Moreover, the semantics approximates any infinite computation.

**Corollary 3.1 (Infinite computations)**

Assume that $d.s \in \exists \hat{x}_{1}(\lbrack P_{1} \rbrack \cap \uparrow(c_{i}.e^{\omega}))$ and that $\langle \hat{x}_{1}; P_{1}; c_{1} \rangle \rightarrow^* \langle \hat{x}_{1}; P_{1}; c_{1} \rangle \rightarrow^* \langle \hat{x}_{n}; P_{n}; c_{n} \rangle \rightarrow^* \cdots$ Then $\bigwedge \exists \hat{x}_{i}(c_{i}) \subseteq d$. □

**Proof**

Recall that procedure calls must be guarded next. Then any infinite behavior in $P_{1}$ is due to a process of the form $(\text{abs } \hat{x}; c)Q$ that executes $Q[i/\hat{x}]$ and adds new information of the form $e[i/\hat{x}]$. By an analysis similar to that of Lemma 4, we can show that $d$ entails $e[i/\hat{x}]$. □

**Example 7 (Infinite behavior)**

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

$$\langle 0 ; P ; c \rangle \rightarrow^* \langle \{x_{1}, \cdots , x_{i} \}; P_{n}; \text{out}(x_{1}) \sqcup \cdots \sqcup \text{out}(x_{i}) \sqcup \text{out}(w) \rangle \rightarrow^*$$

At any step of the computation, the observable store is $\text{out}(w) \sqcup \bigcup_{i \in 1..n} \exists x_{i} \text{out}(x_{i})$, which is equivalent to $\text{out}(w)$. Note also that $\text{out}(w).e^{\omega} \in \lbrack P \rbrack$.

For the converse of Theorem 3, we have similar technical problems as in the case of $\text{tcc}$, namely: the combination of the $\text{local}$ operator with the $\text{unless}$ constructor. Thus, similar to $\text{tcc}$, completeness is verified only for the fragment of $\text{utcc}$ where
there are no occurrences of unless processes in the body of local processes. The reader may refer de Boer et al. (1995) and Nielsen et al. (2002a) for counterexamples showing that \([P] \not\equiv sp(P)\) 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 \(p_i(\tilde{x}) := P_i\). We say that \(\mathcal{D}.P\) is locally independent if for each process of the form (local \(\tilde{x};c\)) \(Q\) in \(P\) and \(P_i\) it holds that (1) \(Q\) does not have occurrences of unless processes; and (2) if \(Q\) calls to \(p_j(\tilde{x})\), then \(P_j\) also satisfies 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 \(\langle \tilde{x}; P; d \rangle \xrightarrow{\ast} \langle \tilde{x}'; P'; d' \rangle \rightarrow\). We shall prove that \(\exists\tilde{x}(d) \cong \exists\tilde{x}'(d')\) and \(s \in \exists \tilde{x}'[[F(P')]\). We proceed by induction on the lexicographical order on the length of the internal derivation \((\rightarrow\ast)\) and the structure of \(P\), where the predominant component is the length of the derivation. The locally independent condition is used for the case \(P = (\text{local} \tilde{x}; c)Q\). 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 \(\langle \tilde{z}; Q; d \rangle \xrightarrow{\ast} \langle \tilde{x}' \cup \tilde{x}' \parallel Q'; d' \parallel \langle \tilde{z}; S; d \rangle \xrightarrow{\ast} \langle \tilde{x}' \cup \tilde{x}' \parallel S'; d'' \parallel e \rangle\). Therefore, assuming by alpha conversion that \(\tilde{x}' \cap \tilde{y} = \emptyset\), \(\exists\tilde{x}'(d) \cong \exists\tilde{y}'(d' \parallel c \parallel e)\) and by rule RPAR,

\[
\langle \tilde{x}; Q \parallel S, d \rangle \xrightarrow{\ast} \langle \tilde{x} \cup \tilde{x}' \cup \tilde{y}' \parallel Q' \parallel S'; d' \parallel d'' \parallel c \parallel e \rangle \rightarrow\]

We note that \(\exists\tilde{x}([P]) \cap ([Q]) = \exists\tilde{x}([P]) \cap ([Q])\) if \(\tilde{x} \cap \text{fv}(Q) = \emptyset\) (see Proposition 7 in Appendix D.). Since \(F(Q') \parallel S' = F(Q') \parallel F(S')\) and \(\tilde{x}' \cap \text{fv}(S') = \tilde{y}' \cap \text{fv}(Q') = \emptyset\), we conclude \(s \in \exists \tilde{x}'[Q', \parallel R')\).

**Case** \(P = (\text{abs} \tilde{x}; c)Q\). We can show that:

\[
\langle \tilde{x}; P; d \rangle \xrightarrow{\ast} \langle \tilde{y}_1; P_1 \parallel Q_1[\tilde{t}_1/\tilde{x}] ; d_1 \rangle \xrightarrow{\ast} \langle \tilde{y}_2; P_2 \parallel Q_2[\tilde{t}_2/\tilde{x}] \parallel Q_2 \parallel Q_2[\tilde{t}_2/\tilde{x}] \parallel Q_2[\tilde{t}_2/\tilde{x}] ; d_2 \parallel d_2 \parallel d_2 \rangle \xrightarrow{\ast} \cdots \xrightarrow{\ast} \langle \tilde{y}_n; P_n \parallel Q_n \parallel \cdots \parallel Q_n \parallel \cdots \parallel Q_n \parallel d_n \parallel \cdots \parallel d_n \rangle
\]

where \(P_n\) takes the form \((\text{abs} \tilde{x}; c; E_n)Q\) and \(E_n = \{d_{\tilde{z}_i}, \ldots, d_{\tilde{z}_i}\}\). In the above derivation \(d_i\) represents the constraint added by \(Q_i[\tilde{t}_i/\tilde{x}]\). Note that \(Q_i[\tilde{t}_i/\tilde{x}] = Q_i[\tilde{t}_i/\tilde{x}]\). There is a derivation (shorter than that for \(P\)) for each \(d_{\tilde{z}_i} \in E_n\) of the form

\[
\langle \tilde{y}_i; Q_i[\tilde{t}_i/\tilde{x}] ; d_i \rangle \xrightarrow{\ast} \langle \tilde{y}_i' ; Q_i[\tilde{t}_i/\tilde{x}] ; d_i^{m_i} \rangle \rightarrow\]

Since \(d.s \in [P]\), by Proposition 1 we know that \(d.s \in [[Q_i[\tilde{t}_i/\tilde{x}]]]\) and by induction, \(\exists\tilde{y}_i(d_i) \cong \exists\tilde{y}_i(d_i^{m_i})\). Furthermore, it must be the case that \(s \in \exists \tilde{y}_i([F(Q_i[\tilde{t}_i/\tilde{x}])])\). Let \(e\)
be the constraint $\exists \bar{y}_n (d_i^{m_i} \sqcup \ldots \sqcup d_n^{m_n})$. Given that $\exists \bar{y}_i (d_i) \cong \exists \bar{y}_i (d_i^{m_i})$, we have $\exists \bar{x}(d) \cong e$. Furthermore, given that $F(P_n) = \text{skip}$:

$$(\text{abs } \bar{x}; c) Q \xrightarrow{(d,e)} (\text{local } \bar{y}_n) F \left( \prod_{d_i \in E_n} Q_i^{m_i}[\bar{t}_i/\bar{x}] \right)$$

Since $s \in \exists \bar{y}([F(Q_i^{m_i}[\bar{t}_i/\bar{x}])])$ for all $d_i \in E_n$, we conclude

$$s \in \exists \bar{y}_n [F \left( \prod_{d_i \in E_n} Q_i^{m_i}[\bar{t}_i/\bar{x}] \right)]$$

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

$$(\dot{y}; (\text{local } \bar{x}) Q; d) \xrightarrow{\text{local } \bar{y}} (\bar{x} \cup \bar{y}; Q; d) \xrightarrow{*} (\dot{y}''; Q'', c) \vdash$$

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

Since $s' \in \exists \bar{y}'[F(Q')]$, $s \in \exists \bar{x} \exists \bar{y}'[F(Q')]$. Nevertheless, note that in the above derivation of $(\text{local } \bar{x}) Q$, the final process is $Q''$ and not $Q$. Since $Q$ is monotonic, there are no $\text{unless}$ 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 $R'[\bar{t}/\bar{x}]$ resulting from a process of the form $(\text{abs } \bar{y}; e) R$ s.t. $d'' \vdash e[\bar{t}/\bar{x}]$ and $c \not\vdash e[\bar{t}/\bar{x}]$. Therefore, by Rule $\text{DPAR}$, it must be also the case that $s' \in [F(Q'')]$ and then $s \in \exists \bar{x}, \bar{y}'[F(Q'')]$. Finally, note that $\bar{y}''$ is not necessarily equal to $\bar{y}'$. With a similar analysis we can show that in $Q'$ there are possibly more $\text{local}$ processes running in parallel than in $Q''$, and then $s \in \exists \bar{y}''[F(Q'')]$. □

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 $w \cong w'$.

Note 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 that we develop in the next section, we require the semantics to be a sound approximation of 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, for example, 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
\[
\mathcal{C} = \langle \mathcal{C}, \sqsubseteq, \sqcup, t, \mathbf{f}, \text{Var}, \exists, D \rangle
\]
\[
\mathcal{A} = \langle \mathcal{A}, \sqsubseteq, \sqcup, t, \mathbf{f}, \text{Var}, \exists, D \rangle
\]
consists of an abstract domain \((\mathcal{A}, \alpha)\) and a surjective and monotonic abstraction function \(\alpha : \mathcal{C} \rightarrow \mathcal{A}\). We lift \(\alpha\) to sequences of constraints in the obvious way.

We shall use \(c_\alpha, d_\alpha\) 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 “\(\alpha\)” when no confusion arises.

The entailment \(\models^\alpha\) is defined as in the concrete counterpart, i.e., \(c_\alpha \sqsubseteq^\alpha d_\alpha\) iff \(d_\alpha \models^\alpha c_\alpha\).

Similarly, \(d_\alpha \models^\alpha c_\alpha\) iff \(d_\alpha \models^\alpha c_\alpha\) and \(d_\alpha \sqsubseteq^\alpha c_\alpha\).

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

**Definition 12 (Correctness)**

Let \(\alpha : \mathcal{C} \rightarrow \mathcal{A}\) be monotonic and surjective. We say that \(\mathcal{A}\) is upper correct w.r.t. the constraint system \(\mathcal{C}\) if for all \(c \in \mathcal{C}\) and \(x, y \in \text{Var}\):

1. \(\alpha(\exists x(c)) \sqsubseteq^\alpha \exists x(\alpha(c)).\)
2. \(\alpha(d_{\sqcup i}) \sqsubseteq^\alpha d_{\sqcup i}^\alpha.\)

Since \(\alpha\) is monotonic, we also have \(\alpha(c \sqcup d) \models^\alpha \alpha(c) \sqcup^\alpha \alpha(d)\).

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 \(\mathcal{C}\) be the Herbrand constraint system. As abstract constraint system \(\mathcal{A}\), let constraints be propositional formulas representing groundness information as in \(x \land (y \leftrightarrow z)\), which means that \(x\) 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.
Abstract interpretation of temporal CCP

Fig. 4. (a) $c'_z$ approximates $c$ (i.e., $c'_z \preceq c$) and $c_z = \alpha(c)$ is the best approximation of $c$ (Definition 13). Since $\alpha$ is monotonic and $c \preceq d$, $c_z \preceq^z d_z$. In (b), assume that for all $d$ s.t. $d \not\vdash c$, $d$ is not approximated by $c_z$. Then all constraint $c'$ approximated by $c_z$ (the upper cone of $c$) entails $c$. In this case $c_z \vdash_A c$ (Definition 14).

Fig. 5. Abstract denotational semantics for utcc $\vdash_A$ and $\triangleright$ are as in Definition 14. $\overline{A}$ denotes the set complement of $A$.

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

Definition 13 (Approximations)
Let $(C, \alpha, A)$ be a description satisfying the conditions in Definition 11. Given $d_z = \alpha(d)$, we say that $d_z$ is the best approximation of $d$. Furthermore, for all $c_z \preceq^z d_z$ we say that $c_z$ approximates $d$ and we write $c_z \preceq d$. This definition is point-wise 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 $\alpha$ of the description $(C, \alpha, A)$. We shall dwell a little upon the description of the rules $A_{\text{ASK}}$ and $A_{\text{UNL}}$. The other cases are self-explanatory.

Given the right abstraction of the synchronization mechanism of blocking ask operator 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 $\text{when } c \text{ do } P$ 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 $d, c \in C$. Note that from $\alpha(d) \vdash^Z \alpha(c)$
we cannot, in general, conclude \( d \vdash c \). Take, for instance, the constraint systems in Example 8. We know that \( x = a \) \( \cong \) \( x = b \) but \( x = a \not\vdash x = b \). Assume now we were to define the abstract semantics of ask processes as:

\[
\mathcal{Q}_X^Z = \mathcal{T}(\overline{a(c)})..\mathcal{A} \cup (\mathcal{T}(\overline{a(c)})..\mathcal{A} \cap \mathcal{Q}_X^Z)
\]

(1)

A correct analysis of the process \( P = \text{tell}(x = a) \parallel \text{when } x = b \text{ do } \text{tell}(y = b) \) should conclude that only \( x \) is definitely Ground. Since \( x = a \) \( \not\vdash x = b \), if we use equation (1), the analysis ends with the result \( (x \land y) ..\mathcal{A} \), i.e., it wrongly concludes that \( x \) and \( y \) are definitely ground.

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

**Definition 14 (\( \vdash_A \) relation)

Let \( d_A \in \mathcal{A} \) and \( c \in \mathcal{C} \). We say that \( d_A \) entails \( c \), notation \( d_A \vdash_A c \) if for all \( c' \in \mathcal{C} \) s.t. \( d_A \not\in c' \) it holds that \( c' \vdash c \). We shall use \( \vdash c \) to denote the set \( \{ d_A \in \mathcal{A} | d_A \vdash_A c \} \).

In words, the (abstract) constraint \( d_A \) entails the (concrete) constraint \( c \) if all constraints approximated by \( d_A \) entail \( c \) (see Figure 4b). Then in equation AASK, we guarantee that if the abstract computation proceeds (i.e., \( d_A \vdash_A c \)), then every concrete computation it approximates proceeds too.

In equations DABS and DLOC we use the operators \( \exists \) and \( \forall \) analogous to those in Notation 5. In this context, these are defined on sequences of constraints in \( \mathcal{A} \) and use the elements \( \exists^\alpha, \sqcup^\alpha \), and \( d_{\exists^\alpha} \) instead of their concrete counterparts:

\[
\exists \check{x}(s_a) = \{ s_x \in \mathcal{A} | \text{there exists } s'_{x} \in S_{x} \text{ s.t. } \exists^\alpha \check{x}(s_x) \equiv \exists^\alpha \check{x}(s'_{x}) \}
\]

\[
\forall \check{x}(s_a) = \{ \exists^\alpha \check{x}(s_x) \in S_{x} | \check{y} \subseteq \text{Var}, s_x \in S_{x} \text{ and for all } s'_{x} \in \mathcal{A} \}
\]

\[
\text{if } \exists^\alpha \check{x}(s_x) \equiv \exists^\alpha \check{x}(s'_{x}), \left( d_{\exists^\alpha} \right)^{\psi} \subseteq s'_{x} \text{ and } \text{adm}(\check{x}, \check{v}) \text{, then } s'_{x} \in S_{x} \}
\]

We omitted the superindex "\( \alpha \)" 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 \( \vdash_A \) 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:

\[
\mathcal{Q}_X^Z = \mathcal{T}(\overline{a(c)})..\mathcal{A} \cup (\mathcal{T}(\overline{a(c)})..\mathcal{A} \cap \mathcal{Q}_X^Z)
\]

(2)

The problem here is that \( \alpha(d) \not\vdash_A c \) does not imply, in general, \( d \not\vdash c \). Take, for instance, \( \alpha \) in Example 8. We know that \( x \not\vdash_A x = [a] \) and \( x = [a] \not\vdash x = [a] \). Now let \( Q = \text{unless } c \text{ next } \text{tell}(e) \), \( d \) be a constraint s.t. \( d \vdash c \) and \( d_A = \alpha(d) \). We know by rule DUNL that \( d . \psi \in \mathcal{Q} \). If \( \alpha(d) \not\vdash_A c \), then by using equation (2), we conclude that \( d_A (d . \psi)^{\psi} \not\in \mathcal{Q}_X^Z \). Hence, we have a sequence \( s \) such that \( s \in \mathcal{Q} \) and \( \alpha(s) \not\in \mathcal{Q}_X^Z \) and the abstract semantics cannot be shown to be a sound approximation of the concrete semantics (see Theorem 5).

Note that defining \( d_A \not\vdash_A c \) as true iff \( c' \not\vdash c \) for all \( c' \) approximated by \( d_A \) does not solve the problem. This is because under this definition \( d_A \not\vdash_A c \) does not hold for any \( d_A \) and \( c \). To see this, note that \( \check{f} \) 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 \( \mathcal{A}^o \) (Rule \( \text{AUNL} \)).

Now we can formally define the abstract semantics as we did in Section 3.

Given a description \( (\mathcal{E}, \alpha, \mathcal{A}) \), we choose as abstract domain is \( A = (A, \sqsubseteq^\mathcal{A}) \) where
\( A = \{ X \mid X \in \mathcal{P}(\mathcal{A}^o) \} \) and \( (\mathcal{E}^2)^o \in X \) and \( X \sqsubseteq^\mathcal{A} Y \) iff \( X \supseteq Y \). The bottom and top
of this domain are similar to the concrete domain, i.e., \( \mathcal{A}^o \) and \( \{(\mathcal{E}^2)^o\} \) respectively.

**Definition 15**
Let \( [[\_]]_X^\mathcal{A} \) be as in Figure 5. The abstract semantics of a program \( D.P \) is defined as
the least fixpoint of the continuous semantic operator:

\[
T_\mathcal{A}^o(X)(p(\vec{t})) = [[Q(\vec{t}/\vec{x})]]_X^\mathcal{A} \text{ if } p(\vec{x}):=Q \in D
\]

We shall use \( [[P]]^\mathcal{A}_X \) to denote \( [[P]]^\mathcal{A}_{T_\mathcal{A}^o} \).

The following proposition shows the monotonicity of \( [[\_]]^\mathcal{A} \) and the continuity of
\( T_\mathcal{A}^o \). The proof is analogous to that of Proposition 2.

**Proposition 3 (Monotonicity of \( [[\_]]^\mathcal{A} \) and continuity of \( T_\mathcal{A}^o \))**
Let \( P \) be a process and \( X_1 \sqsubseteq^\mathcal{A} X_2 \sqsubseteq^\mathcal{A} X_3 \ldots \) be an ascending chain. Then
\( [[P]]^\mathcal{A}_{X_i} \sqsubseteq^\mathcal{A} [[P]]^\mathcal{A}_{X_{i+1}} \) (Monotonicity). Moreover, \( [[P]]^\mathcal{A}_{\bigcup_{i} X_i} = \bigcup_{i} [[P]]^\mathcal{A}_{X_i} \) (Continuity).

### 4.3 Soundness of the approximation

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

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

**Proof**
Let \( A = (A, \sqsubseteq^\mathcal{A}) \), \( E = (E, \sqsubseteq^\mathcal{E}) \) and \( \alpha : E \rightarrow A \) and \( \gamma : A \rightarrow E \) be defined as follows:

\[
\alpha(S) = \{ \beta(s) \mid s \in S \} \text{ for } S \in \{ X \mid X \in \mathcal{P}(\mathcal{E}^o) \text{ and } \mathcal{E}^o \in X \}
\]
\[
\gamma(S) = \{ s \mid \beta(s) \in S \} \text{ for } S \in \{ X \mid X \in \mathcal{P}(\mathcal{A}^o) \text{ and } (\mathcal{E}^2)^o \in X \}
\]

where \( \beta \) is the point-wise extension of \( \alpha' \) over sequences. Note that \( \beta \) is a monotonic
and surjective function between \( \mathcal{E}^o \) and \( \mathcal{A}^o \) and set intersection is the lub in both
\( E \) and \( A \). We conclude by the fact that any additive and surjective function between
complete lattices defines the 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 \( I : \text{ProcHeads} \rightarrow E \),
\( X : \text{ProcHeads} \rightarrow A, \beta \) be as in Proposition 4 and \( p \) be a process definition. Then

\[
\alpha(I(p)) = \{ \beta(s) \mid s \in I(p) \} \quad \gamma(X(p)) = \{ s \mid \beta(s) \in X(p) \}
\]

We conclude here by showing that concrete computations are safely approximated
by the abstract semantics.
Theorem 5 (Soundness of the approximation)
Let \((s, \alpha, \mathcal{A})\) be a description and \(A\) be upper correct w.r.t. \(C\). Given a \texttt{utcc} program \(\mathcal{D}.p\) if \(s \in \llbracket P \rrbracket\), then \(\alpha(s) \in \llbracket P \rrbracket^a\).

Proof
Let \(d_2.s_2 = \alpha(d.s)\) and assume that \(d.s \in \llbracket P \rrbracket\). Then \(d.s \in \llbracket P \rrbracket\), where \(I\) is the least fixpoint of \(T_{\mathcal{D}}\). By the continuity of \(T_{\mathcal{D}}\), there exists \(n\) s.t. \(I = T_{\mathcal{D}}^n(I_\bot)\) (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 \(P = (\text{abs } \vec{x}; c) Q\).** Let \([\vec{i}/\vec{x}]\) be an admissible substitution. We shall prove that \(s \in \llbracket (\text{when } c \text{ do } Q)[\vec{i}/\vec{x}] \rrbracket\) implies \(s_2 \in \llbracket (\text{when } c \text{ do } Q)[\vec{i}/\vec{x}] \rrbracket^a\). The result follows from Proposition 1 and from the fact that \(s_2 \in \forall \vec{\alpha}(\llbracket \text{when } c \text{ do } Q \rrbracket^a)\) iff \(s_2 \in \llbracket (\text{when } c \text{ do } Q)[\vec{i}/\vec{x}] \rrbracket^a\) for all \(\text{adm}(\vec{i}, \vec{x})\). The proof of the previous statement is similar to that of Proposition 1 and it appears in Appendix D.

Assume that \(d \vdash c[\vec{i}/\vec{x}]\). Then \(d.s \in \llbracket Q[\vec{i}/\vec{x}] \rrbracket\) and we distinguish the following two cases:

1. \(d_2 \vdash_{\mathcal{D}} c[\vec{i}/\vec{x}]\). Since \(d.s \in \llbracket Q[\vec{i}/\vec{x}] \rrbracket\), \(d.s \in \exists \vec{\alpha}(\llbracket Q \rrbracket \cap \uparrow (d_{\vec{x}}^a))\). Therefore, there exists \(d'.s'\), an \(\vec{x}\)-variant of \(d.s\), s.t. \(d'.s' \in \llbracket Q \rrbracket\) and \(d'.s' \in \uparrow (d_{\vec{x}}^a)\). By (structural) inductive hypothesis, \(\alpha(d'.s') \in \llbracket Q \rrbracket^a\). Furthermore, by monotonicity of \(\alpha\) and Property (2) in Definition 12, we derive \(\alpha(d'.s') \in \uparrow (d_{\vec{x}}^a)_{\alpha}\). Hence, \(\alpha(d'.s') \in (\llbracket Q \rrbracket^a \cap \uparrow (d_{\vec{x}}^a))\). Since \(\exists \vec{\alpha}(d.s) = \exists \vec{\alpha}(d'.s')\), by Property (1) in Definition 12, we have \(\exists \vec{\alpha}(d.s) = \exists \vec{\alpha}(d'.s')\) (i.e., \(\alpha(d'.s')\) is an \(\vec{x}\)-variant of \(d_2.s_2\)). Then \(d_2.s_2 \in \exists \vec{\alpha}(\llbracket Q \rrbracket^a \cap \uparrow (d_{\vec{x}}^a))\) and we conclude \(d_2.s_2 \in \llbracket Q[\vec{i}/\vec{x}] \rrbracket^a\).

2. \(d_2 \not\vdash_{\mathcal{D}} c[\vec{i}/\vec{x}]\). Hence, trivially \(d_2.s_2 \in \llbracket (\text{when } c \text{ do } Q)[\vec{i}/\vec{x}] \rrbracket^a\).

We conclude by noticing that if \(d \vdash c[\vec{i}/\vec{x}]\), then \(d_2 \not\vdash_{\mathcal{D}} c[\vec{i}/\vec{x}]\) and therefore \(d_2.s_2 \in \llbracket (\text{when } c \text{ do } Q)[\vec{i}/\vec{x}] \rrbracket^a\).

**Case \(P ::= p(\vec{x})\).** Let \(p(\vec{x}) ::= Q\) in \(\mathcal{D}\) be a process definition. If \(d.s \in \llbracket p(\vec{i}) \rrbracket\), then \(d.s \in I(p(\vec{i}))\) (recall that \(I = \text{lf}p(T_{\mathcal{D}}))\). We know that \(d.s \in \llbracket Q[\vec{i}/\vec{x}] \rrbracket\) and then \(d.s \in \llbracket Q[\vec{i}/\vec{x}] \rrbracket\), where \(I' = T_{\mathcal{D}}^m(I_\bot)\) with \(m < n\). By induction, and continuity of \(T_{\mathcal{D}}\), we know that \(d_2.s_2 \in \llbracket Q[\vec{i}/\vec{x}] \rrbracket^a\) and then \(d_2.s_2 \in \llbracket p(\vec{i}) \rrbracket^a\). \(\square\)

### 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 \((\alpha_k, \gamma_k)\) with \(\alpha_k : (\mathcal{A}^\omega, \leq^\mathcal{A}) \rightarrow (\mathcal{A}^\omega_k, \leq^\mathcal{A})\) and \(\gamma_k : (\mathcal{A}^\omega_k, \leq^\mathcal{A}) \rightarrow (\mathcal{A}^\omega, \leq^\mathcal{A})\). As for the function \(\alpha_k\), 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 \(s'(i) = t\) for \(i > k\).

It is easy to see that, for any \(k\), \((\alpha_k, \gamma_k)\) defines the Galois connection between \((\mathcal{A}^\omega, \leq^\mathcal{A})\) and \((\mathcal{A}^\omega_k, \leq^\mathcal{A})\). Thus, it is possible to use the compositions of Galois connections for obtaining a new abstraction (Cousot and Cousot 1992).

If \(\mathcal{A}\) in \((\mathcal{E}, \mathcal{A}, \mathcal{A})\) leads to a Noetherian abstract domain \(\mathfrak{A}\), then the abstraction obtained from the composition of \(\alpha\) and any \(\alpha_k\) 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 the \(k\), 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. (2) Next, 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. (3) Finally, 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 \(\Sigma\) be a signature with constant symbols in \(\mathcal{P} \cup \mathcal{K}\), function symbols \(\text{enc}, \text{pair}\), \(\text{priv}\), and \(\text{pub}\), and predicates \(\text{out}(\cdot)\) and \(\text{secret}(\cdot)\). Constraint in \(\mathcal{C}\) are formulas built from predicates in \(\Sigma\), conjunction (\(\sqcap\)), 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 \(\text{enc}(m, k)\) (encryption) and \(\text{pair}(m_1, m_2)\) (composition). For the generation of keys, \(\text{priv}(k)\) stands for the private key associated with the value \(k\) and \(\text{pub}(k)\) 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:

\[
\begin{align*}
\text{Disam} &: \neg (\text{abs}(x,y) ; \text{out}(x,y)) \text{tell}(\text{out}(x) \sqcup \text{out}(y)) \\
\text{Comp} &: \neg (\text{abs}(x,y) ; \text{out}(x) \sqcup \text{out}(y)) \text{tell}(\text{out}(x)) \\
\text{Enc} &: \neg (\text{abs}(x,y) ; \text{out}(x) \sqcup \text{out}(y)) \text{tell}(\text{out}([x]_{\text{pub}(y)})) \\
\text{Dec} &: \neg (\text{abs}(x,y) ; \text{out}(\text{priv}(y)) \sqcup \text{out}([x]_{\text{pub}(y)})) \text{tell}(\text{out}(x)) \\
\text{Pers} &: \neg (\text{abs}(x) ; \text{out}(x)) \text{next tell}(\text{out}(x)) \\
\text{Spy} &: \text{Disam} \parallel \text{Comp} \parallel \text{Enc} \parallel \text{Dec} \parallel \text{Pers} \parallel \text{next Spy}
\end{align*}
\]

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 out(m) may add messages of unbounded length. Take, for example, the process Comp that will add the constraints out(m), out((m, m, m)), 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); and 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 Song et al. (2001); Armando and Compagna (2008); and Escobar et al. (2011). Roughly, we shall cut the messages generated of length greater than a given \( \kappa \), thus allowing us to model a bounded version of the attacker.

Before defining the abstraction, we note that the constraint system we are considering includes existentially quantified syntactic equations. For these kind of equations it is necessary to refer to their solved forms 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 \( f_\vec{v}(t(\vec{y})) \subseteq \vec{y} \). 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; equations 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 equations 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 \( \Sigma \) in Definition 17. Let \( \lg : \mathcal{M} \to \mathbb{N} \) be defined as \( \lg(m) = 0 \) if \( m \in \mathcal{P} \cup \mathcal{N} \cup \text{Var} \); \( \lg([m_1, m_2]) = 1 + \lg(m_1) + \lg(m_2) \). Let \( \text{cut}_\kappa(m) = m \) if \( \lg(m) \leq \kappa \). Otherwise, \( \text{cut}_\kappa(m) = m_{\rightarrow} \), where \( m_{\rightarrow} \notin \mathcal{M} \) represents all the messages whose length is greater than \( \kappa \). We define \( \alpha(c) \) as \( \alpha_k(NF(c)) \) where

\[
\begin{align*}
\alpha_k(c(m)) &= c(\text{cut}_\kappa(m)) \\
\alpha_k(c \sqcup c') &= \alpha_k(c) \sqcup \alpha_k(c') \\
\alpha_k(\exists c) &= \exists x \alpha_k(c) \\
\alpha_k(d_{\rightarrow}) &= d_{\rightarrow'} \text{ where } t' = \text{cut}_\kappa(t)
\end{align*}
\]

Abstract interpretation of temporal CCP

Fig. 6. Steps of the Needham–Schroeder protocol.

\[ \text{Init}(i, r) \quad \triangleq \quad \text{(local } m) \text{tell(out}(\text{local } m, i) \quad \text{next}(\text{abs } x; \text{out}(\text{local } m, i) \quad \text{tell(out}(\text{local } m, i)) \quad \text{next}(\text{abs } x; \text{out}(\text{local } m, i))) \quad \text{next(Resp(r))}) \]

Fig. 7. utcc model of the Needham–Schröder protocol.

\[ \text{Secrete}(x) \quad \triangleq \quad \text{tell(secret}(x)) \quad \text{next(Resp(r))} \]

\[ \text{SpKn}() \quad \triangleq \quad \text{tell(out}(\text{Abs}(A)) \cup \text{out}(\text{Abs}(B))) \quad \text{next(Resp}(r) \quad \text{Secrete}(n)) \]

and \( NF(c) \) is a solved form of the constraint \( c \). We omit the superscript \( \alpha \) in the abstract operators \( \sqcup, \exists, \) and \( d_{\alpha}^{2} \) to simplify the notation.

We note that the previous abstraction reminds of the depth-\( \kappa \) 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 \( n \) 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 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 \( \text{priv}(C) \), she can decrypt the message \( \text{Out}(\text{Abs}(C)) \) 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 \text{local} constructs and the process \text{tell(out}(m)) models the broadcast of the message \( m \). Inputs (message reception) are modeled by \text{abs} processes as in Example 3. In Resp, we use the process \text{Secrete}(n) to state that the nonce \( n \) cannot be revealed. Finally, the process \text{SpKn}
corresponds to the initial knowledge of the attacker: the names of the principals, their public keys, and the leaked keys in the set $\text{Bad}$ (e.g., the private key of $C$ in the configuration of Figure 6 (b)).

Consider the following process:

$$\text{NS} : = \text{Spy} \parallel \text{SpKn} \parallel \text{Init}(A, C) \parallel \text{Resp}(B)$$

By using the composition of $\alpha_3$ (as in Definition 18) and the sequence abstraction $2$-sequence, we obtain the abstract semantics of NS as shown 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. $s \in \llbracket \text{NS} \rrbracket^\alpha$. Then there exists a $m_1$-$n_1$-variant $s' = c'_1.c'_2$ of $s$ s.t.

$$c'_1 \vdash \text{out}(\{m_1, A\}_{\text{pub}(C)}) \cup \text{out}(\text{priv}(C)) \cup \text{out}(\{m_1, A\}_{\text{pub}(B)})$$
$$c'_2 \vdash \text{out}(\{m_1, n_1, A\}_{\text{pub}(A)}) \cup \text{out}(\{n_1\}_{\text{pub}(C)}) \cup \text{out}(\text{secret}(n_1)) \cup \text{out}(\text{out}(n_1))$$

This means that the nonce $n_1$ appears as a 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 whether a variable will always be bound to a ground term. This information can be used, for example, 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 analysis 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 $\text{gen}_a(x)$ adds an “$a$” to the stream $x$ when the environment provides $\text{go}_a = []$ as input. Under input $\text{stop}_a = []$, $\text{gen}_a(x)$ terminates the stream binding its tail to the empty list. The process $\text{gen}_b$ can be explained similarly. The process $\text{assign}(x, y)$ persistently equates $x$ and $y$. Finally, $\text{append}(x, y, z)$ binds $z$ to the concatenation of $x$ and $y$. 
We shall use \( \text{Pos} \) (Armstrong et al. 1998) as abstract domain for the groundness analysis. In \( \text{Pos} \), positive propositional formulas represent groundness dependencies among variables. For instance, \( z_G(x = [a|b]) = x \) meaning that \( x \) is a ground variable and \( z_G(x = [y|z]) = x \leftrightarrow (y \land z) \) 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., \( x \sqsubseteq (y \land z) \vdash_{z_G} y \).

**Observation 2 (Precision of Pos with respect to synchronization)** Note that \( \text{Pos} \) does not distinguish between the empty list and a list of ground terms: \( d_k = z_G(x = []) = z_G(x = [a]) = x \) and then \( d_k \not\vdash_{d} x = [] \) (see Definition 14). This affects the precision of the analysis. For instance, let \( P = \text{tell}(x = []) \) and \( Q = \text{when} x = [] \text{ do tell}(y = []) \). 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 \( x.\text{true}^G \in \llbracket P \rrbracket^G \) and then the information added by \( \text{tell}(y = []) \) is lost.

We improve the accuracy of the analysis by using the abstract domain defined in Codish and Cousot (1992) to derive information about type dependencies on terms. The abstraction is defined as follows:

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

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

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

**Definition 19 (Groundness-type constraint system)** Let \( A_{GT} = \langle \mathcal{C}, \llbracket x \rrbracket^{2GT}, \sqsubseteq^{2GT}, \sqsubseteq^{2GT}, \mathcal{F}^{2GT}, Var, \exists^{2GT}, d^{2GT} \rangle \). Given \( c \in \mathcal{C} \), \( z_GT(c) = \langle z_G(c), z_T(c) \rangle \). The operations \( \sqsubseteq^{2GT} \) and \( \exists^{2GT} \) correspond to logical conjunction and existential quantification on the components of the tuple and \( d^{2GT} \) is defined as \( z_G(\tilde{x} = i), z_T(\tilde{x} = i) \). Finally, \( \langle c_k, d_k \rangle \llbracket x \rrbracket^{2GT} \langle c'_k, d'_k \rangle \) iff \( c_k \vdash_{z_G} c'_k \) and \( d'_k \vdash_{x_T} d_k \).

Consider Example 9 and the abstraction \( z \) resulting from the composition of \( z_{GT} \) above and \( \text{sequence}_k \). Note that the program makes use of guards of the form \( \exists x', x''(x = [x'|x'']) \) and \( x = [] \). Note also that \( \text{list}(x, x') \vdash_{d} \exists x', x''(x = [x'|x'']) \), and \( \text{nil}(x) \vdash_{d} x = [] \). Roughly speaking, this guarantees that the chosen domain is accurate w.r.t. the ask processes in the program.
\[ [\text{gen}_a(x) \parallel \text{gen}_b(y) \parallel \text{append}(x,y,z)]^\alpha = \exists x_1(\text{GA}_1) \cap \exists y_1(\text{GB}_1) \cap A_1 \quad \text{where} \]

\[
\begin{align*}
GA_1 &= \uparrow (x \leftrightarrow x_1, \text{list}(x, x_1)).A \cap \\
&\{c.s \mid \text{if } c \vdash_A \text{go}_a = [] \text{ then } s \in \exists x_2(\text{GA}_2) \cap \\
&\{c.s \mid \text{if } c \vdash_A \text{stop}_a = [] \text{ then } \langle x_1, \text{nil}(x_1) \rangle^\omega \leq^\alpha c.s \} \\
\ldots \\
GA_\kappa &= \uparrow (x_{\kappa-1} \leftrightarrow x_\kappa, \text{list}(x_{\kappa-1}, x_\kappa)).c \cap \\
&\{c.e \mid \text{if } c \vdash_A \text{stop}_a = [] \text{ then } c^\omega \langle x_\kappa, \text{nil}(x_\kappa) \rangle \cap \\
&\langle x_{\kappa-1}, \text{list}(x_{\kappa-1}, x_\kappa) \rangle \rangle \cap A.\text{A}_\kappa \}} \\
\ldots \\
A_1 &= \{c.s \mid \text{if } c \vdash_A x = [] \text{ then } (d_{\kappa_1}^\omega)^\omega \leq^\alpha c.s \} \cap \\
&\{c.s \mid \text{if } c \vdash_A \exists x', x_2(x = [x' | x_2]) \text{ then } \\
&c.s \in \exists x' \exists x_2 \exists x_3(\uparrow (x \leftrightarrow x_2, \text{list}(x, x_2))^\omega) \cap \\
&\langle x', \text{list}(x_2, x_3) \rangle^\omega \cap A.\text{A}_1 \cap \}
\ldots \\
A_\kappa &= \{c.e \mid \text{if } c \vdash_A x_\kappa = [] \text{ then } d_{\kappa_1}^\omega \leq^\alpha c.e \} \cap \\
&\{c.e \mid \text{if } c \vdash_A \exists x', x_\kappa'((x = [x' | x_\kappa']) \text{ then } \\
&c.e \in \exists x' \exists x_\kappa' \exists x_\kappa''(\uparrow (x \leftrightarrow x_\kappa', \text{list}(x, x_\kappa''))^\omega) \cap \\
&\langle x_\kappa', \text{list}(x, x_\kappa'') \rangle^\omega \cap \}
\]

Fig. 10. Abstract semantics of the process \( P = \text{gen}_a(x) \parallel \text{gen}_b(y) \parallel \text{append}(x,y,z) \). Definitions of \( \text{gen}_a(x) \), \( \text{gen}_b(y) \), and \( \text{append}(x,y,z) \) are given in Example 9. Sets \( \text{GB}_1, \ldots, \text{GB}_\kappa \) are similar to \( \text{GA}_1, \ldots, \text{GA}_\kappa \) and omitted here.

The semantics of the process \( P = \text{gen}_a(x) \parallel \text{gen}_b(y) \parallel \text{append}(x,y,z) \) is depicted in Figure 10. Assume that \( s = c_1.c_2...c_\kappa \in [P]^\omega \). Let \( n \leq \kappa \) and assume that for \( i < n \), \( c_i \vdash_{\omega} \text{go}_a = [] \) and \( c_n \vdash_{\omega} \text{stop}_a = [] \). Since \( s \in [P]^\omega \), we know that \( s \in [\text{gen}_a(x)]^\omega \) and then we can verify that \( c_n \vdash_{\omega} (x, \text{list}(x)) \). Similarly, take \( m \leq \kappa \) and assume that for \( j < m \), \( c_j \vdash_{\omega} \text{go}_b = [] \) and \( c_m \vdash_{\omega} \text{stop}_b = [] \). We can verify that \( c_m \vdash_{\omega} (y, \text{list}(y)) \). Finally, since \( s \in \text{append}(x,y,z) \), we can show that \( c_{\max(n,m)} \vdash_{\omega} (z, \text{list}(z)) \). In words, the process \( P \) binds \( x, y, \) and \( z \) to ground lists whenever the environment provides as input a series of constraints \( \text{go}_a = [] \) (resp. \( \text{go}_b = [] \)) followed by an input \( \text{stop}_a = [] \) (resp. \( \text{stop}_b = [] \)).

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 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, \( \text{on}, \text{off}, \text{closed}, \) and \( \text{open} \) represent the constraints \( \text{on} = [], \text{off} = [], \text{close} = [], \) and \( \text{open} = [] \) and the symbols \text{yes}, \text{no}, \text{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 \( c_1.c_2...c_\kappa \in [[\text{micCtrl}(\text{Error, Button})]]^\omega \) and there exists \( 1 \leq i \leq \kappa \) s.t. \( c_i \vdash_{\omega} (\text{open} = [] \cup \text{on} = []) \), then it must be the case that \( c_1 \vdash_{\omega} (\text{Error, list(\text{Error})}), \text{i.e., Error is a ground variable. This means that the} \)
Abstract interpretation of temporal CCP

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, 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 $\mathcal{S} = \{\bot, \text{ns}\}$ s.t. $\bot \leq \text{ns}$. Given a constraint system $C = \langle \mathcal{C}, \leq, \cup, t, f, \text{Var}, \exists, d \rangle$, the suspension constraint system $S(C)$ is defined as

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

where $\leq^s, \cup^s$ are defined point-wise, $\exists^s_x(\langle c, c' \rangle) = \langle \exists_x c, c' \rangle$, and $d^s_{\xi_i} = \langle d_{\xi_i}, \bot \rangle$. Given a constraint $c \in \mathcal{C}$, we shall use $\hat{c}$ to denote the constraint $\langle c, \bot \rangle$.

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

$$\text{micCtrl}(\text{Error, Signal}) \vdash$$

$${local \text{Error}', \text{Signal}', \text{er}, \text{sl}} \{$$

$${! \text{tell}(\text{Error} = \text{[er | Error']} \cup \text{Signal} = \text{[sl | Signal']})}$$

$${| \text{when on} \cup \text{open do } ! \text{tell}(\text{er} = \text{yes} \cup \text{Error} = \text{[]} \cup \text{sl} = \text{stop})}$$

$${| \text{when off do } (| ! \text{tell}(\text{er} = \text{no}) \cup \text{next} \text{micCtrl}(\text{Error}', \text{Signal'}))}$$

$${| \text{when closed do } (| ! \text{tell}(\text{er} = \text{no}) \cup \text{next} \text{micCtrl}(\text{Error}', \text{Signal'}))}$$

Fig. 11. Model for a microwave controller (see Notation 3 for the definition of $!$).
We then have:  
\[ \hat{P} = \text{when } \hat{a} \text{ do } (\text{tell}(b) \parallel \text{tell}(\langle a, \text{n}s \rangle)) \quad \hat{Q} = \text{when } \hat{c} \text{ do } (\text{tell}(\hat{d}) \parallel \text{tell}(\langle c, \text{n}s \rangle)) \]

We then have:
\[
\begin{align*}
\hat{P} &= \{ \langle t, \uparrow \perp \rangle, \langle b, \text{n}s \rangle, \langle c, \uparrow \perp \rangle, \langle d, \uparrow \perp \rangle, \langle f, \text{n}s \rangle \}, (C \times \mathcal{S})^\omega \\
\hat{Q} &= \{ \langle t, \uparrow \perp \rangle, \langle a, \uparrow \perp \rangle, \langle b, \uparrow \perp \rangle, \langle d, \text{n}s \rangle, \langle f, \text{n}s \rangle \}, (C \times \mathcal{S})^\omega \\
[\hat{P} \parallel \hat{Q}] &= \{ \langle t, \uparrow \perp \rangle, \langle b, \text{n}s \rangle, \langle d, \uparrow \perp \rangle, \langle f, \text{n}s \rangle \}, (C \times \mathcal{S})^\omega
\end{align*}
\]

where \( \langle c, \uparrow \perp \rangle \) is a shorthand for the couple of tuples \( \langle c, \perp \rangle, \langle c, \text{n}s \rangle \). The process \( P \) suspends on input \( c \) (since \( c \not\models a \)) while \( Q \) under input \( c \) outputs \( d \) and it does not suspend. Note that the system \( P \parallel Q \) does not block on input \( b, d \), or \( f \) and it does on input \( t \). Note also that \( \langle c, \perp \rangle, s \not\in [\hat{P} \parallel \hat{Q}] \). This means that in a store \( c \), at least one of the ask processes in \( \hat{P} \parallel \hat{Q} \) is able to proceed. The key idea is that the process \( \text{tell}(\langle c, \text{n}s \rangle) \) in \( \hat{Q} \) ensures that if \( \langle e, e' \rangle \in [\hat{Q}] \) and \( e \models c \), then it must be the case that \( e' = \text{n}s \). 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 \( \hat{P} = \text{when } \hat{a} \text{ do } (\text{tell}(b)) \), \( \hat{Q} = \text{when } \hat{c} \text{ do } (\text{tell}(\hat{d})) \), and \( \hat{R} = \hat{P} \parallel \hat{Q} \) when \( \hat{a} \uplus \hat{c} \) do \( (\text{tell}(a \uplus c \parallel \text{n}s)) \). Therefore,
\[
\begin{align*}
\hat{P} &= \{ \langle t, \uparrow \perp \rangle, \langle b, \text{n}s \rangle, \langle c, \uparrow \perp \rangle, \langle d, \uparrow \perp \rangle, \langle f, \text{n}s \rangle \}, (C \times \mathcal{S})^\omega \\
\hat{Q} &= \{ \langle t, \uparrow \perp \rangle, \langle a, \uparrow \perp \rangle, \langle b, \uparrow \perp \rangle, \langle d, \text{n}s \rangle, \langle f, \text{n}s \rangle \}, (C \times \mathcal{S})^\omega \\
[\hat{P} \parallel \hat{Q}] &= \{ \langle t, \uparrow \perp \rangle, \langle b, \text{n}s \rangle, \langle d, \uparrow \perp \rangle, \langle f, \text{n}s \rangle \}, (C \times \mathcal{S})^\omega
\end{align*}
\]

Hence, we can conclude that only under input \( f \) neither \( P \) nor \( Q \) suspend.

The previous program transformation can be arbitrarily applied to subterms of the form \( P = \bigcup_{i \in I} \text{when } c_i \text{ do } P_i \). Similarly, for verification purposes, a subterm of the form \( P = (\text{abs } x_1; c_1) \parallel \cdots \parallel (\text{abs } x_n; c_n) P_n \) can be replaced by
\[
P' = \hat{P} \parallel \text{when } (\exists x_1 \hat{c}_1 \cup \ldots \cup \exists x_n \hat{c}_n) \text{ do } \text{tell}(\langle c_1 \cup \ldots \cup c_n, \text{n}s \rangle)
\]

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 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:
\[
\begin{align*}
A(x, y) &\colon= (\text{local } m) (\text{tell}(\text{out}(\langle x, y, m \rangle_{\text{pub}(\text{src})}))) \\
S &\colon= (\text{abs } x, y; m; \text{out}(\langle x, y, m \rangle_{\text{pub}(\text{src})})) \text{tell}(\text{out}(\langle x, m \rangle_{\text{pub}(y)})) \parallel \text{next } S() \\
B(y) &\colon= (\text{abs } x, m; \text{out}(\langle x, m \rangle_{\text{pub}(y)})) B_c \\
\text{Protocol:} & \colon= A(x, y) \parallel S() \parallel B(y)
\end{align*}
\]
Abstract interpretation of temporal CCP

\[ [\text{Protocol}]^\alpha = A.\epsilon \cap S.\epsilon \cap B.\epsilon \text{ where } \\
A = \exists m(\exists x, y, m \in \text{pub}(\text{srv})) \\
S = \forall x, y, m \in \{ (d, c) \mid (d, c) \models A.\text{out}(\{x, y, m\}) \text{ and } \langle d, c \rangle \leq^\text{out} \{x, m\} \} \\
B = \forall x, m \in \{ (d, c) \mid (d, c) \models B.\text{out}(\{x, m\}) \text{ and } \langle d, c \rangle \leq^\text{out} \{x, m\} \} \}

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

where \( B_\epsilon = \text{skip} \) 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 above program by replacing all \( c \) with \( \tilde{c} \) and \( B(y) \) with

\[ B'(y) \equiv (\text{abs } x, m; \tilde{\text{out}}(\{x, m\}))(\text{tell}(\text{out}(\{x, m\}, \text{ns}))) \]

Let \( \alpha \) be as in Definition 18. We choose as abstract domain \( A = S(\alpha) \) and consider sequences of length one. In Figure 12 we show the abstract semantics. We note that \( \langle c, \text{ns} \rangle \), where \( c = \exists m(\text{out}(\{x, y, m\}) \cup \text{out}(\{x, m\})) \), is in the semantics \( [[\text{Protocol}]^\alpha \cup \text{out}(\{x, m\})] \) and \( \langle c, \bot \rangle \notin [[\text{Protocol}]^\alpha \cup \text{out}(\{x, m\})] \). 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

\[ S' \equiv (\text{abs } x, y, m; \text{out}(\{x, y, m\})) \text{tell}(\text{out}(\{x, m\}))) | \text{next } S'() \]

where we changed \( \text{tell}(\text{out}(\{x, m\})) \) to \( \text{tell}(\text{out}(\{x, m\})) \). We can verify that \( \langle c, \bot \rangle \in [[\text{Protocol}]^\alpha \cup \text{out}(\{x, m\})] \) where \( c = \exists m(\text{out}(\{x, y, m\}) \cup \text{out}(\{x, m\})) \). This can warn the programmer that there is a mistake in the code.

\[ \text{6 Concluding remarks} \]

Several frameworks and abstract domains for the analysis of logic programs have been defined (see, e.g., Cousot and Cousot 1992; Armstrong et al. 1998; Codish et al. 1999). These works differ from ours since they do not deal with the temporal behavior and synchronization mechanisms present in \texttt{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 \texttt{ntcc} (Nielsen et al. 2002a) programs (a non-deterministic extension of \texttt{tcc}). The framework presented here is more general since it was designed for the static analysis of \texttt{tcc} and \texttt{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 \texttt{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) that 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. First, 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. Second, 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 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 deterministically. The semantics here presented could smoothly be extended to deal with some forms of non-determinism as 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 also plan to perform freeness, suspension, type, and independence analyses among others. It is well known that these kind of analyses have many applications, e.g., for code optimization in compilers, for improving runtime 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 codes.

Acknowledgements.

We thank Frank D. Valencia, François Fages, and Rémy Haemmerlé for insightful discussions on different subjects related to this work. We also thank 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


Abstract interpretation of temporal CCP


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 $\gamma_0 \rightarrow \gamma_1$, $\gamma_0 \rightarrow \gamma_2$, and $\gamma_1 \neq \gamma_2$. Then, there exists $\gamma_3$ such that $\gamma_1 \rightarrow \gamma_3$ and $\gamma_2 \rightarrow \gamma_3$.

Proof
Let $\gamma_0 = (\vec{x}; P; c)$. The proof proceeds by structural induction on $P$. In each case where $\gamma_0$ has two different transitions (up to $\equiv$) $\gamma_0 \rightarrow \gamma_1$ and $\gamma_0 \rightarrow \gamma_2$, one shows the existence of $\gamma_3$ such that $\gamma_1 \rightarrow \gamma_3$ and $\gamma_2 \rightarrow \gamma_3$.

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

For the case $P = Q || R$, we have to consider three cases. Assume that $\gamma_1 \equiv (\vec{x}_1; Q_1 || R, c_1)$ and $\gamma_2 \equiv (\vec{x}_2; Q_2 || R, c_2)$. Let $\gamma'_0 \equiv (\vec{x}; Q; c_0)$, $\gamma'_1 \equiv (\vec{x}_1; Q_1; c_1)$ and $\gamma'_2 \equiv (\vec{x}_2; Q_2; c_2)$. We know by induction that if $\gamma'_0 \rightarrow \gamma'_1$ and $\gamma'_0 \rightarrow \gamma'_2$, then there exists $\gamma'_3 \equiv (\vec{x}_3; Q_3; c_3)$ such that $\gamma'_1 \rightarrow \gamma'_3$ and $\gamma'_2 \rightarrow \gamma'_3$. We conclude by noticing that $\gamma_1 \rightarrow \gamma_3$ and $\gamma_2 \rightarrow \gamma_3$ where $\gamma_3 \equiv (\vec{x}_3; Q_3 || R, c_3)$. 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 $\gamma_0 \equiv (\vec{x}; P; c_0)$ with $P = (\text{abs} \; \vec{y}; c; D) Q$. One can verify that $\gamma_1 \equiv (\vec{x} \cup \vec{x}_1; P_1; c_0)$ where $P_1$ takes the form $(\text{abs} \; \vec{y}; c; D \cup \{d_{\vec{y}_1}\}) Q || Q[\vec{t}_1/\vec{y}]$ and $\gamma_2 \equiv (\vec{x} \cup \vec{x}_2; P_2; c_0)$ where $P_2$ takes the form $(\text{abs} \; \vec{z}; c; D \cup \{d_{\vec{y}_2}\}) Q || Q[\vec{t}_2/\vec{y}]$. From the assumption $\gamma_1 \neq \gamma_2$, it must be the case that $d_{\vec{y}_1} \neq d_{\vec{y}_2}$. By alpha conversion, we assume that $\vec{x}_1 \cap \vec{x}_2 = \emptyset$. Let $\gamma_3 \equiv (\vec{x} \cup \vec{x}_1 \cup \vec{x}_2; P_3; c_0)$ where $P_3 = (\text{abs} \; \vec{y}; c; D \cup \{d_{\vec{y}_1}, d_{\vec{y}_2}\}) Q || Q[\vec{t}_1/\vec{y}] || Q[\vec{t}_2/\vec{y}]$. Clearly $\gamma_1 \rightarrow \gamma_3$ and $\gamma_2 \rightarrow \gamma_3$ as wanted. 

Observation 4 (Finite Traces)
Let $\gamma_1 \rightarrow \cdots \rightarrow \gamma_n \rightarrow$ by a finite internal derivation. The number of possible internal transitions (up to $\equiv$) in any $\gamma_i = (\vec{x}_i; P_i; c_i)$ in the above derivation is finite.

Proof
We proceed by structural induction on $P_i$. The interesting case is the $\text{abs}$ process. Let $Q = (\text{abs} \; \vec{x}; c) P$. Suppose, to obtain a contradiction, that $c_i \vdash c[\vec{t}/\vec{x}]$ for infinitely many $i$ (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 $\gamma_n \rightarrow$. 


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

Proof
Assume that $P \xrightarrow{(c;\exists(d))} (\text{local}\, \bar{x}) F(Q)$, $P \xrightarrow{(c;\exists'(d'))} (\text{local}\, \bar{x}') F(Q')$ and let $\gamma_1 \equiv \langle \emptyset; P; c \rangle$, $\gamma_2 \equiv \langle \emptyset; P; c \rangle$. If $\gamma_1 \not\rightarrow$, then trivially $\gamma_2 \not\rightarrow$, $d \equiv d'$ and $Q \equiv Q'$. Now assume that $\gamma_1 \rightarrow^{*} \gamma_1' \not\rightarrow$ and $\gamma_2 \rightarrow^{*} \gamma_2' \not\rightarrow$ where $\gamma_1' \equiv \langle \bar{x}; Q; d \rangle$ and $\gamma_2' \equiv \langle \bar{x}'; Q'; d' \rangle$. By repeated applications of Lemma 6, we conclude $\gamma_1' \equiv \gamma_2'$ and then, $d \equiv d'$ 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 $s = c_1. c_2. \ldots, s' = c'_1. c'_2. \ldots$ and that $(s, s') \in io(P)$. We then have a derivation of the form:

$$P \equiv P_1 \xrightarrow{(c_1,c'_1)} P_2 \xrightarrow{(c_2,c'_2)} \cdots P_i \xrightarrow{(c_i,c'_i)} P_{i+1} \cdots$$

For $i \geq 1$, we also know that there is an internal derivation of the form $\langle \emptyset; P_i; c_i \rangle \rightarrow^{*} \langle \bar{x}; P_i'; c_i \rangle \not\rightarrow$ where $P_{i+1} = (\text{local}\, \bar{x}) F(P_i')$.

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 $\leq$ be the minimal ordering relation on processes satisfying: (1) skip $\leq P$. (2) If $P \leq Q$ and $P \equiv P'$ and $Q \equiv Q'$ then $P' \leq Q'$. (3) If $P \leq Q$, for every context $C[\cdot]$, $C[P] \leq C[Q]$. Intuitively, $P \leq Q$ 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 $\bar{x}, \bar{x}'$ if $\langle \bar{x}; P; c \rangle \rightarrow^{*} \langle \bar{x}'; P'; c' \rangle \not\rightarrow$ then for every $d \vdash c$ and $Q$ s.t. $P \leq Q$ there $\langle \bar{y}; Q; d \rangle \rightarrow^{*} \langle \bar{y}; Q'; d' \rangle \not\rightarrow$ for some $\bar{y}$ and $Q'$ with $(\text{local}\, \bar{x}') F(P') \leq (\text{local}\, \bar{y}) F(Q')$ and $\exists \bar{y}(d') \vdash \exists \bar{x}'(c')$. This can be proved by
induction on the length of the derivation using the following two properties:

(a) → is monotonic w.r.t. the store, in the sense that, if \(\langle \hat{x}; P; c \rangle \rightarrow \langle \hat{x}'; P'; c' \rangle\), then for every \(d \vdash c\) and \(Q\) s.t. \(P \leq Q\), \(\langle \hat{x}; Q; d \rangle \rightarrow \langle \hat{y}; Q'; d' \rangle\) where \(\exists \hat{y}(d') \vdash \exists \hat{x}(c')\) and \((\text{local } \hat{x}') P' \leq (\text{local } \hat{y}) Q'\).

(b) For every monotonic process \(P\), if \(\langle \hat{x}; P; c \rangle \not\rightarrow\) then for every \(d \vdash c\) and \(Q\) such that \(P \leq Q\) we have either \(\langle \hat{x}; Q; d \rangle \not\rightarrow\) or \(\langle \hat{x}; Q; d \rangle \rightarrow^* \langle \hat{x}'; Q'; d' \rangle \not\rightarrow\) where \(\exists \hat{x}(d' \vdash \exists \hat{x}(d') \not\rightarrow (\text{local } \hat{x}) F(P) \leq (\text{local } \hat{x}) F(Q)\). The restriction to programs which do not contain unless constructs is essential here. □

**Theorem 2**

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

**Proof**

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

**Appendix B. Detailed proofs Section 3**

**Observation 1** (Equality and \(\hat{x}\)-variants)

Let \(S \subseteq \mathcal{G}^a\), \(\hat{x} \subseteq \text{Var}\) and \(s, w\) be \(\hat{x}\)-variants such that \(d^a_{\hat{x}i} \leq s\), \(d^a_{\hat{x}i} \leq w\) and \(\text{adm}(\hat{x}, i)\).

1. \(s \approx w\).
2. \(\exists\hat{x}(s) \in \Psi(\hat{x}(S))\) iff \(s \in \Psi(\hat{x}(S))\).

**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 \(c \cup d_{\hat{x}i} \subseteq c\), \(d \cup d_{\hat{x}i} \subseteq d\) and \(\exists \hat{x}(c \cup d_{\hat{x}i}) \subseteq \exists \hat{x}(c \cup d_{\hat{x}i})\). Hence, \(c[i/\bar{i}] \approx d[i/\bar{i}]\). Since \(c \vdash \exists \hat{x}(c)\), we can show that \(c \vdash \exists \hat{x}(d)\) and then, \(c \vdash d[i/\bar{i}]\). Since \(d[i/\bar{i}] \supset d_{\hat{x}i}\) (Notation 2) we conclude \(c \vdash d\). The “\(d \vdash c\)” side is analogous and we conclude \(c \approx d\).

Property (2) follows directly from the definition of \(\Psi(\cdot)\). □

**Lemma 4**

Let \([\cdot]\) be as in Definition 9. If \(P \xrightarrow{\langle d, d' \rangle} R\) and \(d \approx d'\), then \(d.[R] \subseteq [P]\).

**Proof**

Assume that \(\langle \hat{x}; P; d \rangle \rightarrow^* \langle \hat{x}'; P'; d' \rangle\). We shall prove that \(\exists \hat{x}(d) \approx \exists \hat{x}(d')\). We proceed by induction on the lexicographical order of 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** \(P = \text{skip}\). This case is trivial.

**Case** \(P = \text{tell}(c)\). If \(\langle \hat{x}; \text{tell}(c); d \rangle \rightarrow \langle \hat{x}; \text{skip}; d \rangle\), then it must be the case that \(d \approx d \cup c\) and \(d \vdash c\). We conclude \(\exists \hat{x}(d).[\text{skip}] \subseteq \exists \hat{x}(\text{tell}(c)).\)

**Case** \(P = (\text{local } \hat{x}; c) Q\). Consider the following derivation

\[
\langle \hat{y}; (\text{local } \hat{x}) Q; d \rangle \rightarrow \langle \hat{y} \cup \hat{x}; Q; d \rangle \rightarrow^* \langle \hat{y} \cup \hat{x}; Q'; d' \rangle \rightarrow
\]
where, by alpha-conversion, $\bar{x} \cap \bar{y} = \emptyset$ and $\bar{x} \cap f\bar{v}(d) = \emptyset$. Assume that $\exists \bar{y}(d) \equiv \exists \bar{x}'(d')$. Since the derivation starting from $Q$ is shorter than that starting from $P$, we conclude $\exists \bar{y}(d).\exists \bar{x}, \bar{y}'[F(Q')] \subseteq \exists \bar{x}, \bar{y}[Q]$.

**Case $P = \text{next } Q$**. This case is trivial since $d.\llbracket Q \rrbracket \subseteq \llbracket P \rrbracket$ for any $d$.

**Case $P = \text{unless } c \text{ next } Q$**.

We distinguish two cases: (1) If $d \vdash c$, then we have $\langle \bar{x}; \text{unless } c \text{ next } Q; d \rangle \rightarrow \langle \bar{x}; \text{skip}; d \rangle \not\rightarrow$ and we conclude $\exists \bar{x}(d).\llbracket \text{skip} \rrbracket \subseteq \exists \bar{x}[\text{unless } c \text{ next } P]$. (2), the case when $d \not\vdash c$ is similar to the case of $P = \text{next } Q$. □

**Lemma 5 (Completeness)**

Let $\mathcal{D}.P$ be a locally independent program s.t. $d.s \in \llbracket P \rrbracket$. If $P \xrightarrow{(d,d')} R$, then $d' \equiv d$ and $s \in \llbracket R \rrbracket$.

**Proof**

Assume that $P$ is locally independent, $d.s \in \llbracket P \rrbracket$ and there is a derivation of the form $\langle \bar{x}; P; d \rangle \rightarrow^{*} \langle \bar{x}'; P'; d' \rangle \not\rightarrow$. We shall prove that $\exists \bar{x}(d) \equiv \exists \bar{x}'(d')$ and $s \in \exists \bar{x}'[F(P')]$. We proceed by induction on the lexicographical order on the length of the internal derivation ($\rightarrow^{*}$) and the structure of $P$, where the predominant component is the length of the derivation. The locally independent condition is used for the case $P = (\text{local } \bar{x}; c)Q$. We present here the missing cases in the body of the paper.

**Case skip**. This case is trivial

**Case $P = \text{tell}(c)$**. This case is trivial since it must be the case that $d \vdash c$ and hence $d' \sqcup c \equiv d$.

**Case $P = \text{next } Q$**. This case is trivial since $\langle \bar{x}; P; d \rangle \not\rightarrow$ for any $d$ and $\bar{x}$ and $F(P) = Q$.

**Case $P = \text{unless } c \text{ next } Q$.** If $d \vdash c$ the case is trivial. If $d \not\vdash c$ the case is similar to that of $P = \text{next } Q$.

**Case $P = p(\bar{t})$.** Assume that $p(\bar{x}) : -Q \in \mathcal{D}$. If $d.s \in \llbracket p(\bar{t}) \rrbracket$ then $d.s \in \llbracket Q[\bar{t}/\bar{x}] \rrbracket$. By using the rule $R_{\text{CALL}}$ we can show that there is a derivation

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

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

**Appendix C. Detailed proofs Section 4**

**Theorem 5 (Soundness of the approximation)**

Let $(\mathcal{C}, \mathcal{G}, \mathcal{A})$ be a description and $\text{A}$ be upper correct w.r.t. $\text{C}$. Given a utcc program $\mathcal{D}.P$, if $s \in \llbracket P \rrbracket$ then $\alpha(s) \in \llbracket P \rrbracket^\circ$.

**Proof**

Let $d_s.s_s = \alpha(d.s)$ and assume that $d.s \in \llbracket P \rrbracket$. Then, $d.s \in \llbracket P \rrbracket|_I$, where $I$ is the lfp of $T_{\mathcal{D}}$. By the continuity of $T_{\mathcal{D}}$, there exists $n$ s.t. $I = T^n_{\mathcal{D}}(I_\bot)$ (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 $P = \text{skip}$.** This case is trivial.
Case $P = \text{tell}(c)$. We must have $d \vdash c$ and by monotonicity of $\alpha$, $d_s \vdash \alpha(c)$. We conclude $d_s, s = \{s\}$. 

Case $P = Q \parallel R$. We must have that $s \in \{Q\}$ and $s \in \{R\}$. By inductive hypothesis, we know that $s_s \in \{Q\}$ and $s_s \in \{R\}$ and then, $s_s \in \{Q \parallel R\}$.

Case $P = \{\text{local} \ x \}\{Q\}$. It must be the case that there exists $d' s' \ x$-variant of $d_s$ s.t. $d', s' \in \{Q\}$. Then, by (structural) inductive hypothesis $\alpha(d', s') \in \{Q\}$. We conclude by using the properties of $\alpha$ in Definition 12 to show that $\exists \exists x (\alpha(d,s)) \in \{Q\}$. i.e., $\alpha(d,s)$ and $\alpha(d', s')$ are $\alpha$-variants, and then, $d_s, s_s \in \{Q\}$.

Case $P = \text{next} Q$. We know that $s \in \{Q\}$ and by inductive hypothesis $\alpha(s) \in \{Q\}$. We then conclude $d_s, s_s \in \{Q\}$.

Case $P = \text{unless} c \text{ next} Q$. 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 $x \cap f v(P) = \emptyset$ and let $d.s \in \{P\}$. If $d'.s'$ is an $x$-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 $c(\bar{y})$ and $P(\bar{y})$ to denote constraints and processes where the free variables are exactly $\bar{y}$ and we shall assume that $\bar{y} \cap \bar{x} = \emptyset$. We assume that $d.s \in \{P(\bar{y})\}$ and $d'.s'$ is an $x$-variant of $d.s$. We consider the following cases. The others are easy.

Case $P = \text{when} c(\bar{y})$ do $Q(\bar{y})$. If $d' \vdash c(\bar{y})$ then, by monotonicity, $\exists \exists x (d') \vdash \exists \exists x (c(\bar{y}))$ and then $\exists \exists x (d) \vdash c(\bar{y})$. Hence, it must be the case that $d' \vdash c(\bar{y})$ and $d.s \in \{Q(\bar{y})\}$. By induction we conclude $d'.s' \in \{Q(\bar{y})\}$. If $d' \not\vdash c(\bar{y})$, then $\exists \exists x (d') \vdash c(\bar{y})$ (since $\exists \exists x (d') \subseteq d'$). Hence, $d' \not\vdash c(\bar{y})$ and trivially, $d.s \in \{P\}$ and so $d'.s' \in \{P\}$.

Case $P = \{\text{abs} \ z; c(z, \bar{y})\} Q(\bar{y})$. We know that $d.s \in \{\text{abs} \ z; c(z, \bar{y})\} Q(\bar{y})$. By definition of the operator $\forall (\cdot)$, $\exists \exists x (d.s) \in \{P\}$. Since $\exists \exists x (d'.s') \approx \exists x (d.s)$ we conclude $d'.s' \in \{P\}$. □

Proposition 6
If $\bar{x} \cap f v(P) = \emptyset$ then $\{P\} = \exists \exists x \{P\}$.

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

Proposition 7
If $\bar{x} \notin f v(Q)$ then $\exists \exists x (\{P\} \cap \{Q\}) = \exists \exists x (\{P\}) \cap \{Q\}$.

Proof
(3): Let $d.s \in \exists \exists x (\{P\} \cap \{Q\})$. Then, there exists an $\bar{x}$-variant $d'.s'$ s.t. $d'.s' \in \{P\} \cap \{Q\}$. Then, $d.s \in \exists x (\{P\})$ (by definition) and $d.s \in \{Q\}$ by Proposition 5.

(3): Let $d.s \in \exists \exists x (\{P\}) \cap \{Q\}$. Then, there exists $d'.s'$ $\bar{x}$-variant of $d.s$ s.t. $d'.s' \in \{P\}$. By Proposition 5, $d'.s' \in \{Q\}$ and therefore, $d.s \in \exists \exists x (\{P\} \cap \{Q\})$. □
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 \(s_\alpha\) and \(w_\alpha\) be \(\vec{x}\)-variants such that \((d_{\vec{x}_\alpha})^\omega \leq^2 s_\alpha\), \((d_{\vec{x}_\alpha})^\omega \leq^2 w_\alpha\) and \(\text{adm}(\vec{x}, \vec{t})\). Then \(s_\alpha \cong^2 w_\alpha\).

**Proof**

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

**Proposition 8**

\(s_\alpha \in \forall^\vec{x}(\llbracket P \rrbracket_{\vec{x}})\) if and only if \(s \in \llbracket P[\vec{t}/\vec{x}] \rrbracket_{\vec{x}}\) for all admissible substitution \([\vec{t}/\vec{x}]\).

**Proof**

\((\Rightarrow)\) Let \(s_\alpha \in \forall^\vec{x}(\llbracket P \rrbracket_{\vec{x}})\) and \(s'_\alpha\) be an \(\vec{x}\)-variant of \(s_\alpha\) s.t. \((d_{\vec{x}_\alpha})^\omega \leq^2 s'_\alpha\) where \(\text{adm}(\vec{x}, \vec{t})\). By definition of \(\forall\), we know that \(s'_\alpha \in \llbracket P \rrbracket_{\vec{x}}\). Since \((d_{\vec{x}_\alpha})^\omega \leq^2 s'_\alpha\) then \(s'_\alpha \in \llbracket P \rrbracket_{\vec{x}} \cap \uparrow((d_{\vec{x}_\alpha})^\omega)\). Hence, \(s_\alpha \in \exists^\vec{x}(\llbracket P \rrbracket_{\vec{x}} \cap \uparrow((d_{\vec{x}_\alpha})^\omega))\) and we conclude \(s_\alpha \in \llbracket P[\vec{t}/\vec{x}] \rrbracket_{\vec{x}}\).

\((\Leftarrow)\) Let \([\vec{t}/\vec{x}]\) be an admissible substitution. Suppose, to obtain a contradiction, that \(s_\alpha \in \llbracket P[\vec{t}/\vec{x}] \rrbracket_{\vec{x}}\), there exists \(s'_\alpha\) \(\vec{x}\)-variant of \(s_\alpha\) s.t. \((d_{\vec{x}_\alpha})^\omega \leq^2 s'_\alpha\) and \(s'_\alpha \notin \llbracket P \rrbracket_{\vec{x}}\) (i.e., \(s_\alpha \notin \forall^\vec{x}(\llbracket P \rrbracket_{\vec{x}})\)). Since \(s_\alpha \in \llbracket P[\vec{t}/\vec{x}] \rrbracket_{\vec{x}}\) then \(s_\alpha \in \exists^\vec{x}(\llbracket P \rrbracket_{\vec{x}} \cap \uparrow((d_{\vec{x}_\alpha})^\omega))\). Therefore, there exists \(s''_\alpha\) \(\vec{x}\)-variant of \(s_\alpha\) s.t. \(s''_\alpha \in \llbracket P \rrbracket_{\vec{x}}\) and \((d_{\vec{x}_\alpha})^\omega \leq^2 s''_\alpha\). By Observation 5, \(s'_\alpha \cong^2 s''_\alpha\) and thus, \(s'_\alpha \in \llbracket P \rrbracket_{\vec{x}}\), a contradiction. □