A Framework for Abstract Interpretation of Timed Concurrent Constraint Programs

Moreno Falaschi
Dipartimento di Scienze Matematiche e Informatiche.
Università di Siena.
moreno.falaschi@unisi.it

Carlos Olarte             Catuscia Palamidessi
INRIA and LIX, Ecole Polytechnique
Carlos.olarte,catuscia@polytechnique.fr

Abstract
Timed Concurrent Constraint Programming (tcc) is a declarative model for concurrency offering a logic for specifying reactive systems, i.e. systems that continuously interact with the environment. The universal tcc formalism (utcc) is an extension of tcc with the ability to express mobility. Here mobility is understood as communication of private names as typically done for mobile systems and security protocols. In this paper we consider the denotational semantics for tcc, and we extend it to a "collecting" semantics for utcc based on closure operators over sequences of constraints. Relying on this semantics, we formalize the first general framework for data flow analyses of tcc and utcc programs by abstract interpretation techniques. The concrete and abstract semantics we propose are compositional, thus allowing us to reduce the complexity of data flow analyses. We show that our method is sound and parametric w.r.t. 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, e.g., to perform a groundness analysis for tcc programs. We show the applicability of this analysis in the context of reactive systems. Furthermore, we make also use of the abstract semantics to exhibit a secrecy flaw in a security protocol. We have developed a prototypical implementation of our methodology and we have implemented the abstract domain for security to perform automatically the secrecy analysis.

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

1. Introduction
Concurrent Constraint Programming (ccp) [28] is a process calculus which combines the traditional operational view of process calculi with a declarative one based upon logic. This combination allows ccp to benefit from the large body of reasoning techniques of both process calculi and logic. In fact, ccp-based calculi have successfully been used in the modelling and verification of several concurrent scenarios: biological, security, timed, reactive and stochastic systems, see e.g., [28, 24, 26, 22, 27, 17].

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 (e.g. \( x \leq 42 \)) is parametric in an underlying constraint system.

The tcc model has been extended to consider the execution of processes along a series of time intervals or time-units. In tccp [7], 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) [27], 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 [2] where the system reacts continuously with the environment at a rate controlled by the environment. These languages allow then to program safety critical applications as control systems, for which it is fundamental to develop tools aiming at helping to develop correct, secure, and efficient programs.

Universal tcc [26] (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 \( \pi \)-calculus [21]. Basically, the tcc ask operator when \( e \) do \( P \) is generalized by a parametric ask operator of the form \( (\text{abs } \vec{x}; e) P \) called abstraction. Roughly speaking, an ask process of the form \( P = \text{when } e \text{ do } Q \) remains blocked until the store in strong enough to entail the constraint \( e \) and then \( P \) behaves as \( Q \). In the case of \( P = (\text{abs } \vec{x}; e) Q \), the process \( Q[\vec{t}/\vec{x}] \) is executed for all term \( t \) such that the current store entails \( e[\vec{t}/\vec{x}] \). Notice that when \( \vec{x} \) is the empty vector, we recover the tcc ask operator and thus, all the results we obtain here for utcc programs apply directly to tcc.

Several domains and frameworks, e.g. [6, 5, 1], have been proposed for the analysis of logic programs. The particular characteristics of the timed ccp programs pose additional difficulties for the development of such tools in this language. Namely, the concurrent, timed nature of the language, and the synchronization mechanisms by 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 mobility, particularly, the infinite internal computations generated by the abs operator in utcc.

We develop here a semantics for tcc and utcc which collects all concrete information which is then suitable to properly abstract the properties of interest. This semantics is based on closure op-
Hence we can develop analyses for several applications of utcc and allows us to effectively approximate the operational semantics of utcc and compositionally describe the behavior of programs. We prove this semantics to be fully abstract w.r.t the operational semantics for a significant fragment of the calculus. Next, we propose an abstract denotational semantics which approximates the concrete one.

Our framework is formalized by abstract interpretation techniques and is parametric w.r.t. the abstract domain. It allows us to exploit also 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 both, tcc and utcc (e.g., type systems [16], logical characterizations [20, 22, 26], semantics [27, 25, 22]).

The abstraction we propose proceeds in two-levels. First, we approximate the constraint system leading to an abstract constraint system. We give the general conditions which have to be satisfied for ensuring the soundness of the abstraction. Next, since we are dealing with infinite sequences of (abstract) constraints, we approximate the output of the program by a finite cut. It is worth noticing that the abstract semantics here proposed is computable and compositional. Thus, it allows us to master the complexity of the data-flow analyses. Moreover, the abstraction over-approximates the concrete semantics and then it preserves safety properties.

To the best of our knowledge, we are the first ones to propose a general abstract interpretation framework for a language adhering to the above-mentioned characteristics of tcc or utcc programs. Hence we can develop analyses for several applications of utcc or its sub-calculus tcc (see [24] for a survey of applications of ccf-based languages). In particular, in this paper we instantiate our framework in two different scenarios. The first one tailors an abstract domain for groundness and type dependencies analysis in logic programming to perform a groundness analysis of a abstract domain for groundness and type dependencies analysis in logic programming. Moreover, the abstraction approximates the concrete semantics and then it preserves safety properties.

The second scenario presents an abstract domain for groundness and type dependencies analysis in logic programming to perform a groundness analysis of a abstract domain for groundness and type dependencies analysis in logic programming. Moreover, the abstraction approximates the concrete semantics and then it preserves safety properties.

In [26, 22]) can be seen as an instance of this definition. All results of this paper still hold, of course, when more concrete systems are considered.

A cylindric constraint system is a structure

\[ C = (C, \leq, \cup, \text{true}, \text{false}, \text{Var}, \exists, d) \text{ s.t.:} \]

- \( (\leq, \cup, \text{true}, \text{false}) \) is a lattice with \( \cup \) the lub operation (representing the logical and), and \( \text{true}, \text{false} \) the least and the greatest elements in \( C \) respectively. Elements in \( C \) are called constraints with typical elements \( c, c', d, d' \).

- \( \text{Var} \) is a denumerable set of variables and for each \( x \in \text{Var} \) the function \( \exists \) : \( C \to C \) 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. \( \exists x (c \cup d) = \exists x c \cup \exists x d \).
  4. \( \exists x \exists y c = \exists y \exists x c \).

- For each \( x, y \in \text{Var} \), \( d_{xy} \in C \) is a diagonal element and it satisfies: (1) \( d_{xy} = \text{true} \).
  (2) If \( x \) is different from \( y \) then \( d_{xy} = \exists x(d_{xx} \cup d_{yx}) \).
  (3) If \( x \) is different from \( y \) then \( c \leq d_{xy} \cup \exists x(c \cup d_{xy}) \).

The cylindrification operators model a sort of existential quantification, helpful for defining the hiding operator as we explain below. The diagonal elements are useful to model parameter passing in procedures calls. If \( C \) contains an equality theory, then \( d_{xy} \) can be thought as the formulae \( x = y \).

We say that \( d \) entails \( c \) in \( C \) iff \( c \leq d \) and we write \( d \vdash c \). If \( d \vdash c \) and \( e \vdash d \) we write \( d \equiv e \).

We lift the previous notions to sequences of constraints. We denote respectively by \( C^n, C^\omega \) the set of finite and infinite sequences of constraints with typical elements \( s, s', \ldots \). We use \( c^n \) to denote the sequence \( c, c, \ldots \). The length of \( s \) is denoted by \( |s| \) and the empty sequence by \( \epsilon \). The \( i \)-th element in \( s \) is \( s(i) \). We write \( s \leq s' \) iff \( |s| \leq |s'| \) and for all \( i \in \{1, \ldots, |s|\} \), \( s(i) \vdash s'(i) \). If \( |s| = |s'| \) and for all \( i \in \{1, \ldots, |s|\} \), \( s(i) \equiv s'(i) \), we shall write \( s \equiv s' \).

We denote by \( T \) the set of terms in the constraint system. We use \( \bar{\delta} \) for a sequence of terms \( t_1, \ldots, t_n \) with length \( |\bar{\delta}| = n \). If \( |\bar{\delta}| = 0 \) then \( \bar{\delta} \) is written as \( e \). We use \( e[\bar{\delta}/\bar{x}] \), where \( |\bar{\delta}| = |\bar{x}| \) and \( x \)'s are pairwise distinct, to denote \( c \) in which the free occurrences of \( x \) have been replaced with \( t_i \). The substitution \( e[\bar{\delta}/\bar{x}] \) will be similarly applied to other syntactic entities.

2.1 Reactive Systems and Timed CCP

Reactive systems [2] are those that react continuously with their environment at a rate controlled by the environment. For example, a controller or a signal-processing system, receive a stimulus (input) from the environment. It computes an output and then, waits for the next interaction with the environment.

In the ccf model, the shared store of constraints grows monotonically, i.e., agents cannot drop information (constraints) from it. Then, a simple reactive systems that changes a signal (the value of a variable) cannot be modeled: The conjunction of the constraints \( \text{signal} = \text{on} \) and \( \text{signal} = \text{off} \) leads to an inconsistent store.

The timed ccf calculus (tcc) [27] extends ccf for reactive systems. Time is conceptually divided into time intervals (or time units). In a particular time interval, a ccf process \( P \) gets an input \( a \) from the environment, executes with this input as the initial store, and when it reaches its resting point, it outputs the resulting store \( d \) to the environment. The resting point determines also a residual process \( Q \) which is then executed in the next time unit. The resulting store \( d \) is not automatically transferred to the next time unit. This way, computations during a time-unit proceed monotonically but outputs of two different time-units are not supposed to be related to each other. Therefore, the variable \( \text{signal} \) above may change its value when passing from one time-unit to the next one.

In the following we present the syntax of tcc following the notation in [22].
micCtrl(Error, Button) :-
  (local E', B', e, b)
  !tell Error = [e [E'] | Button = [b | B']
  \ when on | \ open do
  !tell e = yes \ E = [] \ b = stop
  \ when off do tell e = no \ next micCtrl E', B'
  \ when closed do tell e = no \ next micCtrl E', B'

Figure 1. tcc model for a microwave controller.

DEFINITION 1 (tcc Processes). The set Proc of tcc processes is built from constraints in the underlying constraint system by the following syntax:

\[ P, Q := \text{skip} \mid \text{tell}(c) \mid \text{when } c \do P \mid P \parallel Q \mid (\text{local } \vec{x}; c) P \mid \text{next } P \mid \text{unless } \neg c \next P \mid \not! P \mid p(\vec{x}) \]

The process \( \text{skip} \) does nothing thus representing inaction. The process \( \text{tell}(c) \) adds \( c \) to the store in the current time interval making it available to the other processes. The ask process \( \text{when } c \do P \) remains blocked until the store is strong enough to entail the guard \( c \); if so, it behaves like \( P \).

The parallel composition of \( P \) and \( Q \) is denoted by \( P \parallel Q \). Given a set of indexes \( I = \{i_{1}, ..., i_{n}\} \), we shall use \( \prod_{i \in I} P_{i} \) to denote the parallel composition \( P_{1} \parallel ... \parallel P_{n} \).

The process \( (\text{local } \vec{x}; c) P \) binds \( \vec{x} \) in \( P \) by declaring it private to \( P \). It behaves like \( P \) except that all the information on the variables \( \vec{x} \) produced by \( P \) can only be seen by \( P \) and the information on the global variable in \( \vec{x} \) produced by other processes cannot be seen by \( P \). The local information on \( \vec{x} \) produced by \( P \) corresponds to the constraint \( c \) representing a local store. When \( c = \text{true} \), we shall simply write \( (\text{local } \vec{x}) P \) instead of \( (\text{local } \vec{x}; \text{true}) P \).

We shall use \( \text{bv}(Q) \) (resp. \( \text{fv}(Q) \)) to denote the set of bound (resp. free) variables occurring in \( Q \).

The unit-delay \( \text{next } P \) executes \( P \) in the next time unit. The time-out unless \( \neg c \next P \) is also a unit-delay, but \( P \) is executed in the next time unit iff \( \neg c \) is not entailed by the final store at the current time interval. We use \( \text{next}^{n} P \) as short for \( \text{next} \ldots \text{next } P \) with \( \text{next} \) repeated \( n \) times. Finally, the replication \( !P \parallel P \parallel \text{next}^{2} P \parallel ... \parallel \text{next}^{n} P \) (i.e., unboundedly many copies of \( P \) but one at a time.

Assume a (recursive) procedure definition \( p(\vec{y}) : \rightarrow P \) where \( \text{fv}(P) \subseteq \vec{y} \). The call \( p(\vec{x}) \) replaces the formal parameters \( \vec{y} \) in \( P \) with the actual parameters \( \vec{x} \). Recursive calls in \( P \) must be guarded by a \( \text{next} \) process to avoid non-terminating sequences of recursive calls during a time-unit (see [27, 22]).

Let us give an example of a control system modeled in tcc.

EXAMPLE 1 (Control System). Assume a simple control system for a microwave checking that the door must be closed when it is turned on. Otherwise, it must emit an error signal. The specification in tcc of this system is depicted in Figure 1.

In this tcc program, constraints of the form \( X = [e | X'] \) asserts that \( X \) is a list with head \( e \) and tail \( X' \). This way, the process micCtrl binds Error to a list ended by “yes” when the microwave was turned on and the door was open at the same interval of time. Furthermore, the constant stop is added into the list Button signaling the environment that the microwave must be powered off.

Later on, in Section 5.2, we shall show how the abstract interpretation framework developed here allows for the verification of the system above.

2.2 Mobile behavior and UTCC

The tcc calculus lacks of mechanisms for name passing, i.e., mobility in the sense of the \( \pi \)-calculus [21]. Let us illustrate this with an example. Let \( \text{out}(\cdot) \) be a constraint and let \( P = \text{when } \text{out}(x) \do R \) a system that must react when receiving a stimulus of the form \( \text{out}(n) \) for \( n > 0 \). We notice that under input \( \text{out}(42) \), \( P \) does not execute \( R \) since \( \text{out}(42) \) does not entail \( \text{out}(x) \) (i.e. \( \text{out}(42) \not\rightarrow \text{out}(x) \)). The issue here is that \( x \) is a free-variable and hence 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 [26], tcc is extended for mobile reactive systems leading to universal timed csp (utcc). To model mobile behavior, utcc replaces the tcc ask operation when \( c \do P \) with a more general 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, \( Q = (\text{abs } \vec{x}; c) P \) performs \( P[\vec{x}/\vec{e}] \) in the current time interval for all the terms \( \vec{e} \ such that \( c(\vec{e}/\vec{x}) \) is entailed by the current store. For example, \( P = (\text{abs } \vec{x}; \text{out}(x)) R \) under input \( \text{out}(42) \) executes \( R[\text{out}(42)/\vec{x}] \).

From a programming point of view, we can then see the variables \( \vec{x} \) in the abstraction \( (\text{abs } \vec{x}; c) P \) as the formal parameters of \( P \) (see Remark 1).

DEFINITION 2 (utcc Processes). The utcc processes result from replacing in the syntax in Definition 1 the expression \( c \do P \) with \( (\text{abs } \vec{x}; c) P \) with the variables in \( \vec{x} \) being pairwise distinct.

As explained above, the process \( Q = (\text{abs } \vec{x}; c) P \) executes \( P[\vec{x}/\vec{e}] \) in the current time interval for all the terms \( \vec{e} \ such that \( c(\vec{e}/\vec{x}) \) is entailed by the store. When \( [\vec{x}] = 0 \) (i.e. \( \vec{x} = c \)), we recover the tcc ask operator and we write \( c \do P \) instead of \( (\text{abs } \vec{x}; c) P \).

The process \( Q = (\text{abs } \vec{x}; c) P \) binds \( \vec{x} \) in \( P \) and \( c \). Therefore, we extend accordingly the sets \( \text{bv}(Q) \) and \( \text{fv}(Q) \) of bound and free variables. Furthermore \( Q \) evolves into \( \text{skip} \) at the end of the time unit, i.e. abstractions are not persistent when passing from one time-unit to the next one.

DEFINITION 3 (utcc programs). Given a set of procedure declarations \( D \), a utcc program takes the form \( D.P \) where \( P \) is an utcc process (Definition 2). For every procedure name, we assume that there exists one and only one corresponding declaration in \( D \).

REMARK 1. The utcc calculus was introduced in [26] without procedure definitions. Here we add them to properly deal with tcc programs with recursion. In utcc, recursive definitions do not add any expressiveness since they can be encoded by using abstractions. See Appendix B to see this encoding.

We conclude this section with an example of mobile behavior in utcc. Here, a process \( P \) sends a local variable to \( Q \). Then, both processes can communicate through the shared variable.

EXAMPLE 2. Assume two components \( P \) and \( Q \) of a system such that \( P \) creates a local variable that must share with \( Q \). Roughly, this system can be modeled as

\[
P = (\text{local } \vec{x}; \text{out}(x) \parallel P')
\]
\[
Q = (\text{abs } \vec{z}; \text{out}(z)) Q'
\]

In the next section, we shall see that the parallel composition of \( P \) and \( Q \) evolves to a process of the form

\[
(\text{local } \vec{x}; (P' \parallel Q'[\vec{x}/\vec{z}]))
\]
where $P'$ and $Q'$ share the local variable $x$ created by $P$. Then, any information produced by $P'$ on $x$ can be seen by $Q'$ and vice versa.

### 2.3 Operational Semantics (SOS)

The structural operational semantics (SOS) of tcc and utcc considers transitions between process-store configurations $(P,c)$ with stores represented as constraints and processes quotiented by $\equiv$. We use $\gamma, \gamma', \ldots$ to range over configurations.

**Definition 4.** Let $\equiv$ be the smallest congruence satisfying: (0) $\equiv \equiv Q$ if they differ only by a renaming of bound variables $(1)\ P \parallel \text{skip} \equiv P$ $(2)\ P \parallel Q \equiv Q \parallel P$ $(3)\ P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$ $(4)\ P \parallel (\text{local} \bar{x}; c) \equiv (\text{local} \bar{x}; c) \parallel (P \parallel Q)$ if $\bar{x} \not\in \text{fo}(P)$ $(5)\ (\text{local} \bar{x}; c) \text{skip} \equiv \text{skip}$. Extend $\equiv$ by decreeing that $(P, c) \equiv (Q, c)$ if $P \equiv Q$.

Transitions are given by the relations $\longrightarrow$ and $\longrightarrow^*$ in Table 1. The internal transition $(P, d) \longrightarrow (P', d')$ should be read as “$P$ with store $d$ reduces, in one internal step, to $P'$ with store $d'$.” The observable transition $P \overset{(c,d)}{\longrightarrow} R$ 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.

We only describe some of the rules in Table 1. See [22, 26] for further details. All the rules are standard for tcc and they are easily seen to realize the operational intuitions given above. The only new rule is that for $(\text{abs} \bar{x}; c) P$, which allows to define utcc. As clarified below, the seemingly missing rules for next and unless processes are given by $\text{Robs}$.

Let $Q = (\text{local} \bar{x}; c) P$ in Rule $\text{Rloc}$. The global store is $d$ and the local store is $c$. We distinguish between the external (corresponding to $Q$) and the internal point of view (corresponding to $P$). From the internal point of view, the information about $x$, possibly appearing in the “global” store $d$, cannot be observed. Thus, before reducing $P$ we first hide the information about $x$ that $Q$ may have in $d$ by using the cylinderification operator $\exists_x d$. Similarly, from the external point of view, the observable information about $x$ that the reduction of the internal agent $P$ may produce (i.e., $c'$) cannot be observed. Thus we also hide it by $\exists_x c'$. Before adding it to the global store. Additionally, we make $c'$ the new private store of the evolution of the internal process.

Let $Q = (\text{abs} \bar{x}; c) P$ in Rule $\text{Rabs}$. If the current store entails $c[\bar{d}]/\bar{x}$ then $P[\bar{d}]/\bar{x}$ is executed. Additionally, the abstraction persists in the current time interval to allow other potential replacements of $\bar{x}$ in $P$ but $c$ is augmented with $\bar{x} \not\in \bar{t}$ to avoid executing $P[\bar{d}]/\bar{x}$ again. We assume then the constraint “$\not\exists$” to be defined in the constraint system.

The rule $\text{Rcall}$ makes use of the diagonal elements (see Section 2) to model parameter passing as simply done in ecp [28]. In this equation,

\[ \Delta^x_P = (\text{local} \bar{a}) (\text{local} \bar{y}) (\text{tell}(d_{a\bar{a}}) \parallel \text{tell}(d_{\bar{a}\bar{y}}) \parallel P) \]

where the variables in $\bar{a}$ are assumed to occur neither in the declaration nor in the process $P$, and $d_{\bar{a}\bar{y}}$ denotes the constraint $\bigcup_{i \in \mathbb{C}} \Delta^x_P d_{a\bar{a}b}$. Roughly speaking, $\Delta^x_P$ equates the formal parameters $\bar{x}$ and the actual parameters $\bar{y}$.

Rule $\text{Robs}$ says that an observable transition from $P$ labeled with $(c, d)$ is obtained from a terminating sequence of internal transitions from $(P, c)$ to $(Q, d)$. The process $R$ to be executed in the next time interval is equivalent to $F(Q)$ (the “future” of $Q$). $F(Q)$ is obtained by removing from $Q$ abstractions and any local information which has been stored in $Q$, and by “unfolding” the sub-terms within next and unless expressions.

Now we can show how the evolution of the processes in Example 2 leads to a configuration where the variable $x$ created by $P$ is sent to $Q$ and then, both processes can communicate using it.

**Example 3.** Let $P$ and $Q$ be as in Example 2. The parallel composition $R = P \parallel Q$ under input true evolves as follows:

\[ (\langle R, \text{true} \rangle \longrightarrow^* \langle (\text{local} \bar{x}; c) (P' \parallel (\text{abs} z; \text{out}(z)) Q'), \exists_y (c) \rangle \longrightarrow^* \langle (\text{local} \bar{x}; c) (P' \parallel Q'[\bar{x}/\bar{z}] \parallel (Q''), \exists_y (c) \rangle \]

where $Q'' = (\text{abs} z; \text{out}(z) \land x \neq z) Q'$ and $c = \text{out}(x)$. Notice $Q$ and $Q'[\bar{x}/\bar{z}]$ share the local variable $x$.

### 2.3.1 Observables and Input-output Behavior

In this section we formally define the behavior of a process $P$ relating its outputs under the influence of a sequence of inputs (constraints) from the environment.

**Definition 5 (IO Behavior).** Let $s = c_1, c_2, \ldots, c_i$ and $s' = c_1', c_2', \ldots, c_i'$ be sequences of constraints. If $P = P_1 (c_1, c_2) P_2 (c_2, c_3) \cdots P_{i-1} (c_{i-1}, c_i) \Rightarrow P_i (c_i, c_i') \Rightarrow P$, we write $P \Rightarrow (s, s')$. The set

\[ \text{io}(P) = \{ (s, s') \mid P \Rightarrow (s, s') \} \]

denotes the input-output behavior of $P$.

In [26], the outputs of a utcc process were proven to be equivalent up to $\equiv$.

**Theorem 1 (Determinism [26]).** Let $P$ be a utcc process. If $P \Rightarrow (s, s')$ and $P \Rightarrow (s, \tilde{s}')$ then $s' \equiv \tilde{s}'$.

Notice that, unlike the other constructs in utcc, the unless operator exhibits non-monotonic input-output behavior in the following sense: Let $P = \text{unless } c \text{ next } Q$. Given $s \leq s'$, if $(s, w), (s', w') \in \text{io}(P)$, it may be the case that $w \not\equiv w'$. For example, take $Q = \text{tell}(d), s = \text{true}$ and $s' = c. \text{true}$. Then, $s = \text{true}. d. \text{true}$ and $w' = c. \text{true}$ with $w \not\equiv w'$.

We then define a monotonic process as follows:

**Definition 6 (Monotonic Processes).** We say that $P$ is a monotonic process if $P$ does not have occurrences of processes of the form unless $c$ next $Q$.

### 2.3.2 Strongest Postcondition

Given a process $P$, we can show that $\text{io}(P)$ is a partial closure operator [25], i.e., it is a function satisfying extensiveness and idempotence. Furthermore, if $P$ is monotonic, $\text{io}(P)$ is a closure operator satisfying additionally monotonicity.

A pleasant property of closure operators is that they are uniquely determined by their set of fixed points, here called the strongest postcondition:

**Definition 7 (Strongest Postcondition).** Given a utcc process $P$, the strongest postcondition of $P$, denoted by $sp(P)$, is defined as the set $\{ s \mid (s, s) \in \text{io}(P) \}$.

Intuitively, $s \in sp(P)$, iff $P$ under input $s$ cannot add any information whatsoever, i.e. $s$ is a quiescent sequent for $P$. We also can 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 the set of sequences (outputs) satisfying the property $A$.

Finally, it is worth noticing that for the monotonic fragment of utcc, the input-output behavior can be retrieved from the strongest postcondition. This is formalized in the following theorem whose proof is standard, given that $\text{io}(\cdot)$ is a closure operator.
Internal Transitions:

\[
\begin{align*}
R_{\text{TELL}} & \quad \langle \text{tell}(c), d \rangle \rightarrow \langle \text{skip}, d \uplus c \rangle \\
R_{\text{RUNL}} & \quad d \vdash c \quad \langle \text{skip}, d \rangle \rightarrow \langle \text{skip}, d \rangle \\
R_{\text{STR}} & \quad \gamma_1 \rightarrow \gamma_2' \quad \frac{\gamma_1 \equiv \gamma_1' \land \gamma_2 \equiv \gamma_2'}{}
\end{align*}
\]

Observable Transition:

\[
\begin{align*}
R_{\text{OBS}} & \quad (P,c) \rightarrow^* (Q,d) \quad \rightarrow \\
R_{\text{PAR}} & \quad (P,c) \rightarrow (P',d) \\
R_{\text{LOC}} & \quad (P, c \uplus (3 \exists d)) \rightarrow (P', c' \uplus (3 \exists d)) \\
R_{\text{REP}} & \quad (P, d) \rightarrow (P \| \text{next } P, d) \\
R_{\text{ABS}} & \quad (\text{abs } c, d) \rightarrow (P \| \text{abs } c, d) \\
R_{\text{CALL}} & \quad p(\bar{d}) : P \in D \\
F(P) & = \begin{cases} 
\text{skip} & \text{if } P = \text{skip} \text{ or } P = (\text{abs } e, c) Q \\
F(P_1) \| F(P_2) & \text{if } P = P_1 \| P_2 \\
(\text{local } \bar{x}) F(Q) & \text{if } P = (\text{local } e, c) Q \\
Q & \text{if } P = \text{next } Q \text{ or } P = \text{unless } c \text{ next } Q
\end{cases}
\]

Table 1. Operational Semantics for tcc and utcc. \( \equiv \) is given in Definition 4. In \( R_{\text{ABS}} \), \( \bar{x} \neq \bar{t} \) denotes \( \bigwedge_{1 \leq i \leq |\bar{x}|} x_i \neq t_i \). If \( |\bar{x}| = 0, \bar{x} \neq \bar{t} \) is defined as \( \text{false} \).

THEOREM 2. Let \( \min \) be the minimum function w.r.t. the order induced by \( \preceq \). Given a monotonic utcc process \( P \), \( (s,s') \in \sigma(P) \) if \( s' = \min(sP \cap \{w \mid s \preceq w\}) \)

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 we develop in Section 4.

Our semantics is built on the closure operator semantics for ccp and tcc in [28, 27] and specifies compositionally the strongest postcondition relation in Definition 7. Unlike the semantics in [25], 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 formulae. 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 procedure definition \( p(\bar{x}) : P \in D \). Such an interpretation is given in terms of the set of the quiescent sequences for \( p(\bar{x}) \).

3.1 Compositional Semantics

Let \( \text{ProcHeads} \) denote the set of process names with their formal parameters and recall that \( \mathcal{C}^* \) stands for the set of infinite sequences of constraints. We shall call \( \text{Interpretation} \) the set of functions in the domain \( \text{ProcHeads} \rightarrow \mathcal{P}(\mathcal{C}^*) \). The semantics is defined as a function \( [\cdot] : (\text{ProcHeads} \rightarrow \mathcal{P}(\mathcal{C}^*)) \rightarrow (\text{Proc} \rightarrow \mathcal{P}(\mathcal{C}^*)) \) which given an interpretation \( J \), associates to each process a set of sequences of constraints.

Let us give some intuitions about the semantic equations in Table 2. Recall that \( [\cdot] \) aims at capturing the strongest postcondition (or quiescent sequences) of a process \( P \), i.e. the sequences \( s \) s.t. \( P \) under input \( s \) cannot add any information whatever. So, \( \text{skip} \) cannot add any information to any sequence (Equation \( D_{\text{SKIP}} \)). The sequences to which \( \text{tell}(c) \) cannot add information are those whose first element entails \( c \) (Equation \( D_{\text{TELL}} \)). A sequence is quiescent for \( P \) if it is for \( P \) and \( Q \) (Equation \( D_{\text{PAR}} \)).

The process \( \text{next } P \) has no influence on the first element of a sequence, thus \( d.s \) is quiescent for it if \( s \) is quiescent for \( P \) (Equation \( D_{\text{NEXT}} \)). A similar explanation can be given for the process \( \text{unless } c \text{ next } P \) (Equation \( D_{\text{UNL}} \)). A sequence \( s \) is quiescent for \( ! P \) if it is quiescent for every process of the form \( \text{next }^n P \) with \( n > 0 \). Then, every suffix of \( s \) must be quiescent for \( P \) (Equation \( D_{\text{REP}} \)).

We say that \( s \) is an \( \bar{x} \)-variant of \( s' \) if \( \exists x(i) = \exists x'(i) \) for \( i > 0 \) (i.e. \( s' \) and \( s \) differ only on the information about \( \bar{x} \)). A sequence \( s \) is quiescent for \( Q = (\text{local } e, c) P \) if there exists an \( \bar{x} \)-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 \( s \).

As we intuitively explained in Section 2.2, the abstraction process can be seen as a parallel composition:

\[ (\text{abs } c; e) P \equiv \prod_{t \in T[P]} (\text{when } c \text{ do } P)[\bar{t}/\bar{x}] \]

where \( T \) denotes the set of terms in the underlying constraint system. Therefore, the Equation \( D_{\text{ABS}} \) is given in terms of the equation for the ask operator [27]; a sequence \( d.s \) is quiescent for \( \text{when } c \text{ do } P \) either if \( d \) does not entail \( c \) or if \( d \) entails \( c \) and \( d.s \) is quiescent for \( P \) (Equation \( D_{\text{ASK}} \)).

Finally, the meaning of a procedure call is directly given by the interpretation \( I \).

The domain of the denotation is \( \mathcal{E} = (E, \subseteq^E) \) where \( E = \mathcal{P}(\mathcal{C}^*) \) and \( \subseteq^E \) is a Smyth-like ordering defined as follows: Let \( X,Y \subseteq E \) and \( \subseteq^E \) be the preorder s.t. \( X \subseteq Y \) iff for all \( y \in Y \), there exists \( x \in X \) s.t. \( x \leq y \). \( Y \subseteq^E X \) if \( X \subseteq Y \) and \( Y \subseteq X \) implies \( Y \subseteq X \). The bottom of \( \mathcal{E} \) is then \( \mathcal{C}^* \) (the set of all the sequences)
and the top is the singleton \{false\} (since false is the greatest element in \(\langle C, \leq \rangle\)).

Formally, the semantics is defined as follows:

**Definition 8 (Concrete Semantics).** Let \(I\) be defined as in Table 2. The semantics of a program \(D,P\) is defined as the least fixed point of the continuous operator:

\[
T_D(I)(p(\cdot)) = [\Delta^\alpha_{D} P][\cdot], \text{ if } p(\cdot) := P' \in D
\]

We shall use \([P]\) to represent \([p(\cdot)]_{p(\cdot) \in T_D}\).

Let us exemplify the least fixed point construction above with a system similar to that of Example 2.

**Example 4.** 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 \(D\) be the following procedure definitions

\[
D = p() \rightarrow -\text{tell}(\text{out}_{a}(x)) \parallel \text{next tell}(\text{out}_{b}(y))
q() \rightarrow (\text{abs } z; \text{out}_{a}(z)) \text{ tell}(\text{out}_{b}(z)) \parallel \text{next q()}
\]

The procedure \(p()\) outputs on channel a the variables x and y in the first and second time-units respectively. The procedure \(q()\) reverts on channel b every message received on channel a. Starting from the bottom interpretation \(I_\omega\) (assigning \(C^\omega\) to each name procedure), the semantics of \(r(\cdot)\) is obtained as follows

\[
I_\omega \quad p \rightarrow C^\omega, q \rightarrow C^\omega, r \rightarrow C^\omega \cap C^\omega = C^\omega
I_1 \quad p \rightarrow \{c.c.| s \mid c \vdash \text{out}_{a}(x)\} \text{ and } c' \vdash \text{out}_{b}(y)\}
q \rightarrow \{c_1.s \mid c_1 \vdash \text{out}_{a}(t)\}\implies c_1 \vdash \text{out}_{b}(t)\}
\]

\[
I_2 \quad p \rightarrow I_1(p)
q \rightarrow \{c_1,c_2,s \mid c_1[1,2] \vdash \text{out}_{a}(t)\}\implies c_1[1,2] \vdash \text{out}_{b}(t)\}
\]

\[
\vdots
\]

\[
I_n \quad p \rightarrow I_{n-1}(p)
q \rightarrow \{s \mid s(i) \vdash \text{out}_{a}(t)\}\implies s(i) \vdash \text{out}_{b}(t)\} \quad \text{ for } i > 0
\]

\[
\tau \rightarrow I_{n-1}(p) \cap I_{n}(q)
\]

where \(t\) denotes any term and \(c_1[1,2] \vdash d\) stands for \(c_1 \vdash d\) and \(c_2 \vdash d\). In words, if \(s \in [r()]\) then \(s(1) \vdash \text{out}_{a}(x)\), \(s(2) \vdash \text{out}_{a}(y)\) and for \(i \geq 1\), if \(s(i) \vdash \text{out}_{a}(t)\) then \(s(i) \vdash \text{out}_{b}(t)\).

3.2 Semantic Correspondence

In this section we prove the semantic correspondence between the operational and the denotational semantics. Before that, it is worth noticing that unlike tcc, some utcc processes may exhibit infinite behavior during a time-unit due to the abstraction operator. Take for example a process of the form \(P = (\text{abs } z; c(x)) \text{ tell}(c(x + 1))\).

Under input c(1), this process will generate constraints of the form \(c(2), c(3), \ldots\), thus never producing an observable transition (recall that observable transitions are obtained from a finite sequence of internal transitions). This behavior will arise in the application to security in Section 5.3, where the model of the attacker may generate infinitely many messages. We shall show later that the abstract semantics allows us to restrict the number of messages generated, thus avoiding this situation.

Considering this fact, it may be the case that sequences in the input-output behavior (and then in the strongest postcondition) are finite or even the empty sequence \(\varepsilon\). Nonetheless, this is not the case for all utcc process. We shall call well-terminated the processes which do not exhibit infinite internal behavior:

**Definition 9 (Well-termination).** The process \(P\) is said to be well-terminated iff for every infinite sequence of constraints \(s\), there exists \(s'\) such that \((s,s') \in \text{io}(P)\). Notice that any tcc process is well-terminated.

The fragment of well-terminated processes is a meaningful one. For instance, it was shown to be enough to encode Turing-powerful formalisms in [25]. It has also found application, e.g., in multimedia interaction systems [23] and declarative interpretation of languages for structured communications [18].

The following theorem shows that if a (finite) sequence \(s\) is in the strongest postcondition, then there exists an infinite sequence \(s'\) in the denotation such that \(s\) is a prefix of \(s'\).

**Theorem 3 (Soundness).** Let \([\cdot]\) be as in Definition 8. Given a program \(D,P\), if \(s \in \text{sp}(P)\) then there exists \(s'\) s.t. \(s,s' \in [P]\). (proof sketch in Appendix A).

For the converse of the previous theorem, we have similar technical problems as in the case of \(tcc\) namely: the combination between the \(local\) and the \(unless\) operator—see [8, 22] for details. Thus, similarly to [8, 22], completeness is achieved only for the following fragment of utcc:

**Definition 10 (Loc. Ind. \& abs-unless fragment).** We say that a process \(P\) is a locally independent (resp. abstracted-unless free) iff \(P\) has no occurrences of \(unless\) processes under the scope of a \(local\) (resp. \(abs\)) operator. These definitions are extended to programs \(D,P\) by decreeing that \(P\) and all \(P_i\) in \(p_i(\cdot)\) : \(P_i \in D\) satisfy the conditions above.

**Theorem 4 (Completeness).** Let \(D,P\) be a locally independent and abstracted-unless free program s.t. \(s \in [P]\). For all prefixes \(s'\) of \(s\), if there exists \(s''\) s.t. \((s', s'') \in \text{io}(P)\) then \(s'' \equiv s'\), i.e., \(s' \in \text{sp}(P)\). (proof sketch in Appendix A).

4. Abstract Interpretation Framework

In this section we develop an abstract interpretation framework [6] for the analysis of utcc programs. The framework is based on the above denotational semantics, thus allowing for a compositional analysis of utcc (and then tcc) programs. The abstraction proceeds in two levels: (1) we abstract the constraint system and then (2) we abstract the infinite sequences of abstract constraints by a finite cut. The abstraction in (1) allows us to reuse the most popular abstract domains previously defined for logic programming. Adapting those domains, it is possible to perform, e.g., groundness, freeness, type and suspension analyses of tcc and utcc programs. Furthermore, it allows us to restrict the set of terms to be considered in the Equation \(D_{\text{Abs}}\). Thus, we can even approximate the output of a non-well-terminated process as we show in Section 5.3. 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 [12] and [31].

**Definition 11.** Given two constraint systems

\[
C = \langle C, \subseteq, \text{true}^\alpha, \text{Var}, \exists, d^\alpha \rangle
A = \langle A, \subseteq^\alpha, \text{true}^\alpha, \text{false}^\alpha, \text{Var}, \exists^\alpha, d^\alpha \rangle
\]

a description \((C, \alpha, A)\) consists of an abstract domain \((A, \subseteq^\alpha)\) and a monotonic abstraction function \(\alpha : C \rightarrow A\). We lift \(\alpha\) to sequences of constraints in the obvious way:

We shall use \(c_s, d_s\) to range over constraints in \(A\) and \(s_s, s'_s\) to range over sequences in \(A^\omega\) and \(A^*\). Let \(i \vdash a\) be defined as in the concrete counterpart, i.e. \(c_i \leq a\) iff \(d_i \vdash^a c_i\). The set of abstract terms is denoted by \(T_s\) and ranged by \(t_s, t'_s, \ldots\).

Following standard lines in [12, 31] we impose the following restrictions over \(\alpha\):
Definition 12. (Correctness). Let $\alpha : C \rightarrow A$ be monotonic. We say that $A$ is upper correct w.r.t. the constraint system $C$ if for all $c \in C$ and $x, y \in V$: (1) $\alpha(3c, c) = \exists x^{\alpha}(c)$, (2) $\alpha(d_{xy}) = d_{xy}$, (3) $\alpha(c \cup d) \leq^D \alpha(c) \cup^D \alpha(d)$. Let $\alpha_t : T \rightarrow T_o$ be the term-abstraction structurally based on $\alpha$. Given the sequence of variables $\vec{x}$ and $\vec{t} \in T^{[\vec{x}]}$, $\alpha_t(\vec{t}) = \alpha_t(\vec{t}^\alpha)$ whenever $\alpha_t(\vec{t}) = \alpha_t(\vec{t}^\alpha)$.

Conditions (1), (2) and (3) relate the cylindrification, diagonal and $\cup$ operators of both constraints systems. Condition (4) is only necessary to have a safe approximation of the $abs$ operator in utcc, but it is not required when analyzing tcc programs. It formally says that substituting for terms mapped to the same abstract term, must lead to the same abstract constraint.

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 5.1.

Example 5. Let the Herbrand constraint system (Hcs) [28] be the concrete domain. In Hcs, a first-order language $L$ with equality is assumed. The entailment relation is that one expects from equality, e.g., $[x = y] = [a] = [a]$ must entail $x = a$ and $y = z$. Terms, as usual, are variables, constants and functions applied on terms. As abstract constraint system, let constraints be predicates of the form $(\mathit{iff}(x, \[\alpha]))$ meaning that $x$ is a ground variable. Abstract terms are variables and the special term $g$ meaning “ground”. In this setting, $\alpha(x = [a]) = \mathit{iff}(x, \[\alpha])$ (i.e., $x$ is a ground variable). Furthermore $\alpha_t(a) = \alpha_t(b) = g$. Therefore, by Condition (4) in Definition 12, $\alpha_t((x = [y])/[a]/[b]) = \alpha_t((x = [y])/[b]/[y]) = \mathit{iff}(x, \[\alpha])$.

We conclude this section by defining when an “abstract” constraint approximates a concrete one.

Definition 13 (Approximations). Let $A$ be upper correct w.r.t. $C$ and $(C, \alpha, A)$ be a description. Given $d_\alpha = \alpha(d)$, we say that $d_\alpha$ is the best approximation of $d$. Furthermore, for all $c_\alpha \leq^A d_\alpha$ we say that $c_\alpha$ approximates $d$ and we write $c_\alpha \approx d$. This definition is extended to sequences of constraints in the obvious way.

4.2 Abstract Semantics

Starting from the semantics in Section 3, we develop here an abstract semantics which approximates the observable behavior of a program and is adequate for modular data-flow analysis. We focus our attention on a special class of abstract interpretations obtained from what we call a sequence abstraction mapping possibly infinite sequences of (abstract) constraints into finite ones.

Definition 14 (Sequence Abstraction). A sequence abstraction $\tau : A^\omega \cup A^* \rightarrow A^*$ is an anti-extensive ($\tau(s_{\alpha}) \leq^A s$) and monotonic operator. We lift $\tau$ to sets of sequences in the obvious way: $\tau(S) = \{s_{\alpha} | s_{\alpha} = \tau(s_{\alpha}) \text{ and } s_{\alpha} \in S\}$.

A simple albeit useful instance of the abstraction $\tau$ is the $sequence(k)$ cut. This abstraction approximates a sequence by projecting it to its first $k$ elements, e.g., $sequence(2)(c_1, c_2, c_3, \ldots) = c_1, c_2$.

Given a description $(C, \alpha, A)$, we choose as concrete domain $E = (E, \mathbb{C})$ as defined in Section 3. The abstract domain is $A = (A, \leq^A)$ where $A = P(A^*)$ and $\leq^A$ is defined similarly to $\mathbb{C}$. Let $X, Y \in A$ and $\leq^A$ be the preorder s.t. $X \leq^A Y$ iff for all $y \in Y$, there exists $x \in X$ s.t. $x \leq^A y$. $X \leq^A Y$ iff $X \subseteq^A Y$ and $Y \subseteq^A X$ implies $Y \subseteq X$. The top and bottom of this domain are, similar to the concrete domain, $A^*$ and $\{false^A, false^A, \ldots\}$ respectively.

We require $A$ to be noetherian (i.e., there are no infinite ascending chains). This guarantees that the fixed point of the abstract semantics can be reached in a finite number of iterations.

The semantic equations are given in Table 3. We shall dwell a little upon the description of the rules $\text{ASK}$, $\text{ABS}$ and $\text{UNL}$. The other cases are easy. For the case of $\text{ASK}$, we follow [31, 11, 12] for the abstract semantics of the synchronization (ask) operator in ccp. Intuitively, the Equation $\text{ASK}$ says that if the abstract computation proceeds, then every concrete computation it approximates proceeds too. This is formalized by the relation $d_\alpha \models \varnothing_{\varnothing} c$, meaning that the abstract constraint $d_\alpha$ entails $c$ if all concrete constraint approximated by $d_\alpha$ entails $c$. 

<table>
<thead>
<tr>
<th>Rule</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{D SKIP}$</td>
<td>$[\text{skip}]_I = C^\omega$</td>
</tr>
<tr>
<td>$\text{D TELL}$</td>
<td>$[\text{tell}(c)]_I = {d.s \in C^\omega</td>
</tr>
<tr>
<td>$\text{D PAR}$</td>
<td>$[P \parallel Q]_I = [P]_I \cap [Q]_I$</td>
</tr>
<tr>
<td>$\text{D NEXT}$</td>
<td>$[\text{next } P]_I = {d.s \in C^\omega</td>
</tr>
<tr>
<td>$\text{D UNL}$</td>
<td>$[\text{unless } c \text{ next } P]_I = {d.s \in C^\omega</td>
</tr>
<tr>
<td>$\text{D REP}$</td>
<td>$[\text{! } P]_I = {s \in C^\omega</td>
</tr>
<tr>
<td>$\text{D LOC}$</td>
<td>$[(\text{local } \vec{x}; c)]_I = {s \in C^\omega</td>
</tr>
<tr>
<td>$\text{D ASK}$</td>
<td>$[(\text{when } c \text{ do } P)]_I = {d.s \in C^\omega</td>
</tr>
<tr>
<td>$\text{D ABS}$</td>
<td>$[(\text{abs } \vec{x}; c)]<em>I = \bigcap</em>{t \in T_c} {[(\text{when } c \text{ do } P)[\vec{t}/\vec{x}]]_I}$</td>
</tr>
<tr>
<td>$\text{D CALL}$</td>
<td>$[p(\vec{x})]_I = I(p(\vec{x}))$</td>
</tr>
</tbody>
</table>

Table 2. Semantic Equations for tcc and utcc constructs. In $D_{\text{ABS}}$, if $|\vec{z}| = 0$ then $T^{[\vec{z}]}$ is defined as $\{\epsilon\}$.
DEFINITION 15. Given $d_a \in A$ and $c \in C$, $d_a \vdash_A c$ iff for all $c' \in C$ s.t. $d_a \propto c'$, $c' \vdash c$.

In Equation $A_{\text{ABSTRACT}}$, we compute the intersection over the abstract terms ($T_n$) and we substitute $\bar{x}$ for a concrete term $t'$ s.t. $\alpha_t(t') = \bar{t}$. Notice that it may be the case that there exists $t_1$, $t_2$ s.t. $\alpha_1(t_1) = \alpha_2(t_2) = \bar{t}$. Using property (4) in Definition 12, we can show that the choice of the concrete term is irrelevant (see Proposition 2 in Appendix A).

One could think of defining the abstract semantics of the unless operator similarly to that of the operator as follows:

$$\text{unless } c \text{ next } P[X] = \tau(\{d_a, s_n | d_a \vdash_A c \text{ and } s_n \in [P[X]]\} \cup \tau(\{d_a, s_n | d_a \vdash_A c\})$$

Nevertheless, this equation leads to a non-safe approximation of the concrete semantics. This is because from $d_a \vdash_A c$, we cannot conclude that $d \vdash c$ where $\alpha(d) = d_a$. To see this, take $Q = \text{unless } c \text{ next } P$ and $d = \text{true}$. Then, $d \vdash c$ and $d \vdash c$. Take $c'$ s.t. $c' \vdash c$ and $c' \vdash c$. Notice that $d_a \vdash_A c'$. If $P \neq \text{skip}$, we have $d_a \vdash_A \text{true}$.

Defining $d_a \vdash_A c$ as true iff $c' \vdash c$ for all $c'$ approximated by $d_a$, does not solve the problem. This is because under this definition, $d_a \vdash_A c$ would not hold for any $d_a$ and $c$: $\text{false}$ entails all the concrete constraints and it is approximated for every abstract constraint.

Therefore, we cannot give a better (safe) approximation of the semantics of $Q = \text{unless } c \text{ next } P$ than $\tau(A^*)$, i.e. $[Q[X] = \text{skip} \bar{X} \text{Rule AUNL}$.}

We define formally the abstract semantics as follows:

DEFINITION 16. Let $[X]$ be as in Table 3. The abstract semantics of a program $D$. $P$ is defined as the least fixpoint of the following continuous semantic operator:

$$T^n_n(X)(p(\bar{x})) = ([\Delta_n^P])_X \text{ if } p(\bar{y}) := P' \in \mathcal{D}$$

We shall use $[P]^T$ to denote $[P]_{\downarrow}(T^n_n)$.

4.3 Soundness of the Approximation

This section proves the correctness of the abstract semantics in Definition 16. We first establish a Galois insertion between the concrete and the abstract domains. From [31, Proposition 3], we deduce the following:

$$\alpha(E) := \tau(\{\alpha(s) | s \in E\})$$

$$\gamma(A) := \{s | \tau(\alpha(s)) \in A\}$$

We have used $\gamma$ to avoid confusion with $\alpha$ in ($C, \alpha, A$). We can lift in the standard way to abstract interpretations [6] the approximation induced by the above abstraction. Let $I : \text{ProcHeads } \rightarrow E, X : \text{ProcHeads } \rightarrow \text{A and } p$ a procedure name. Then

$$\alpha(I)(p) := \tau(\{\alpha(s) | s \in I(p)\})$$

$$\gamma(X)(p) := \{s | \tau(\alpha(s)) \in X(p)\}$$

The following theorem states that concrete computations are safely approximated by the abstract semantics.

THEOREM 5 (Soundness of the approximation). Let $A$ be upper correct w.r.t. $C, (C, \alpha, A)$ be a description and $\tau$ be a sequence abstraction. Let $\{\}$ and $\{\}$ be respectively as in Definitions 8 and 16. Given a $\text{utcc}$ program $D, P$, if $s \in [P]^T$ then $\tau(\alpha(s)) \in [P]^T$. (See proof sketch in Appendix A).

5. Applications

This section describes two specific abstract domains as instances of our framework. Firstly, we tailor two abstract domains from logic programming to perform a groundness and a type analysis of a $\text{tcc}$ program. Then we apply this analysis in the verification of a reactive system in $\text{tcc}$. Secondly, we abstract a constraint system dealing with cryptographic primitives. Here we use the abstract semantics to exhibit a secrecy flaw in a security protocol programmed in $\text{tcc}$.

5.1 Groundness Analysis

In logic programming, one useful analysis is groundness. It aims at determining if a variable will always be bound to a ground term. This information can be used, e.g., for optimization in the compiler (to remove code for suspension checks at runtime) or as base for other data flow analyses such as independence analysis, suspension analysis, etc. Here we present a groundness analysis for a $\text{tcc}$ program. To this end, we shall use as concrete domain the Herbrand constraint system ($\text{Hcs}$) [28] (see Example 5).

Assume the following procedure definitions:

$$\text{gen}_s(x) := (\text{local } x' \text{!tell}(x = [a|x'|]) \text{ when } go_s \text{ do next } \text{gen}_s(x') \text{ when } stop_s \text{ do } \text{tell}(x' = []))$$

$$\text{gen}_c(x) := (\text{local } x' \text{!tell}(x = [b|x'|]) \text{ when } go_c \text{ do next } \text{gen}_c(x') \text{ when } stop_c \text{ do } \text{tell}(x' = []))$$

$$\text{append}(x, y, z) := \text{when } x = [] \text{ do next } \text{append}(y, z) \text{ when } \exists x', y'(x = [x'|x']) \text{ do } \text{tell}(x = [x'|x']) \text{ when } \text{tell}(x = [x'|x']) \text{ next } \text{append}(y, z)$$

The process $\text{gen}_s(x)$ adds to the stream $x$ an “a” when the environment provides $go_s$ as input. Under input $stop_s$, it terminates the stream binding its tail to the empty list. Let $x, go_s$ and $x, stop_s$ be two distinct variables different from $x$ and $x'$, and $go_s$ and $stop_s$ be the constraints $x, go_s = []$ and $x, stop_s = [].$ The process $\text{gen}_s$ can be explained similarly. The process $\text{append}(x, y, z)$ binds $z$ to the concatenation of $x$ and $y$.

We shall use $[1]$ as abstract domain for the groundness analysis. In $\text{Pos}$, positive propositional formulae represent groundness dependencies among variables. Elements in the domain are ordered by logical implication. Let $\alpha_g$ be defined over equations in normal form as:

$$\alpha_g(x = t) = \text{iff}(x, \text{fv}(t))$$

For instance, $\alpha_g(x = [y|z]) = \text{iff}(x, \{y, z\})$ representing the propositional formula $x \Leftrightarrow (y \land z)$ meaning, $x$ is ground if and only if $y$ and $z$ are grounds.

Notice that $\text{Pos}$ does not distinguish between the empty list and a list of ground terms, i.e., $d_a = \alpha_g(x = []) = \alpha_g(x = [a]) = \text{iff}(x, \{\})$. Therefore, we have $d_a \vdash_A x = []$ (see Definition 15).

We can improve the accuracy of our analysis by using the abstract domain in [4] to derive information about type dependencies on terms. The abstraction is defined as follows:

$$\alpha_T(x = t) = \{\text{list}(x, y) \text{ if } t = [y|x], \text{nil}(x) \text{ if } t = []\}$$

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

Let $\text{A}_{\text{sl}} = (A, \leq^\alpha, \text{true}^\alpha, \text{false}^\alpha)$. Var, $\exists^\alpha, \forall^\alpha$ be the abstract constraint system obtained from the reduced product (6) of the previous abstract domains. Elements $g, g' \in A$ are tuples $(c_a, d_a)$ where $c_a$ corresponds to groundness information and $d_a$ to type dependency information. The abstraction function is defined as expected, i.e., $\alpha(c) = g = (\alpha_g(c), \alpha_T(c))$. The operations $\exists^\alpha, \forall^\alpha$ correspond to logical conjunction and existential quantification over the components of the tuple. The diagonal element $d_{\alpha\alpha}$ cor-
responds to $\langle x = y, x = y \rangle$. Finally, $\langle x_0, x_2 \rangle \leq \langle x_2', x_0' \rangle$ if $x_0' \Rightarrow x_2$ and $x_2' \Rightarrow x_0$.

Let $\tau = \text{sequence}(\kappa)$ and $g_1, g_2, ..., g_6 \in \{\text{Gen}_{\kappa}(x)\}^\tau$. By a derivation similar to that of Example 4, if there exists $i \in \{1, ..., \kappa\}$ such that $g_i \vdash \text{stop}_n$, one can show that there exists $\vec{x} = \vec{x}_0, \vec{x}_1, ..., \vec{x}_i$ such that

$$g_i \vdash (x, x_0') \land \bigwedge_{0 \leq j < i} \text{iff}(x_0'_{j + 1}) \land \text{iff}(x_1') \land \{\}$$

Thus, if $g_i \vdash \text{stop}_n$ we can deduce that $x$ is a list and $x$ is a ground variable.

Let $s_k = \{\text{Gen}_{\kappa}(x) \mid \text{Gen}_{\kappa}(y) \mid \text{append}(x, y, z)\}^\tau$. If there exist $i, j \leq \kappa$, $s_k(i) \vdash \text{stop}_p$ and $s_k(j) \vdash \text{stop}_p$, we can show that for $l \geq \max(i, j)$, the variables $x, y$, and $z$ are list of ground variables. More precisely, $s_k(l) \vdash (x, \{\}) \land \text{iff}(y, \{\}) \land \text{iff}(z, \{\})$.

5.2 Analysis of Reactive Systems

Synchronous data flow languages [2] such as Estrel and Lustre can be encoded as tcc processes [30, 27]. This makes tcc an expressive declarative framework for the modeling and verification of reactive systems. Here we show how our framework can provide additional reasoning techniques in tcc for the verification of such systems.

We shall use the groundness analysis developed in Section 5.1 to verify if the simplified version of a control system for a microwave in Example 1 complies with its intended behavior: the door must verify if the simplified version of a control system for a microwave.

Thus, if $g \vdash x$ we can deduce that $x$ is a list and $x$ is a ground variable.

5.3 Analyzing Secrecy Properties

In [26] it was shown that the ability of utcc to model mobile behavior, as in Example 2, allows for the modeling of security protocols. Nevertheless, the model of the attacker is a well-defined process that produces simultaneously many internal reductions. In this section we show how a suitable abstraction of the cryptographic constraint system in [26] may allow us to bound the number of state changes to be considered in a secrecy analysis. Then we exhibit a well-defined flaw in a security protocol.

We consider a constraint system whose terms are the possible expressions generated during the execution of the protocol. Cryptographic primitives are represented as functions over such terms.

**Definition.** Let $\Sigma$ be a signature with constant symbols in $\mathcal{P} \cup \mathcal{K}$, function symbols enc, pair, priv and predicates out(·) and secret(·). Constraints in $\mathcal{C}$ are first-order formulae built over $\Sigma$.

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

As standardly done in the verifications of security protocols, a Dolev-Yao attacker [10] is presupposed, able to eavesdrop, disassemble, compose, encrypt and decrypt messages with available keys. The attacker can be modeled as follows:

**Table 3.** Abstract denotational semantics for $\text{utcc}, \vdash_\Delta$ in Definition 15

<table>
<thead>
<tr>
<th>$\text{A skip}$</th>
<th>$[\text{skip}]_X^\Delta = \tau(A)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{A tell}$</td>
<td>$[\text{tell}(c)]<em>X^\Delta = \tau({d</em>{\alpha,s_{\alpha}} \in A^* \mid d_{\alpha} \vdash \alpha(c)})$</td>
</tr>
<tr>
<td>$\text{A par}$</td>
<td>$[P \parallel Q]_X^\Delta = [P]_X^\Delta \cap [Q]_X^\Delta$</td>
</tr>
<tr>
<td>$\text{A next}$</td>
<td>$[\text{next } P]<em>X^\Delta = \tau({d</em>{\alpha,s_{\alpha}} \in A^* \mid s_{\alpha} \in [P]_X^\Delta})$</td>
</tr>
<tr>
<td>$\text{A unl}$</td>
<td>$[\text{unless } c \quad \text{next } P]_X^\Delta = \tau(A^*)$</td>
</tr>
<tr>
<td>$\text{A rep}$</td>
<td>$[[P]<em>X^\Delta = \tau({s</em>{\alpha} \in A^* \mid \forall \alpha \forall {s_{\alpha} \in [P]_X^\Delta}})$</td>
</tr>
<tr>
<td>$\text{A loc}$</td>
<td>$[[\text{local } \vec{x}; c)]<em>X^\Delta = \tau({s</em>{\alpha} \in A^* \mid \exists s_{\alpha}' \in [\text{local } \vec{x}; c)]_X^\Delta}$</td>
</tr>
<tr>
<td>$\text{A ask}$</td>
<td>$[[\text{when } c \quad \text{do } P]<em>X^\Delta = \tau({d</em>{\alpha,s_{\alpha}} \in A^* \mid d_{\alpha} \vdash \alpha(c) \land d_{\alpha,s_{\alpha}} \in [P]_X^\Delta})$</td>
</tr>
<tr>
<td>$\text{A abs}$</td>
<td>$[[\text{abs } \vec{x}; c)]<em>X^\Delta = \bigcap</em>{c_{\alpha} \in C} [[(\text{when } c \quad \text{do } P)]_X^\Delta]]_X^\Delta$</td>
</tr>
<tr>
<td>$\text{A call}$</td>
<td>$[[p(\vec{x})]_X^\Delta = X(p(\vec{x}))$</td>
</tr>
</tbody>
</table>
The first four processes represent the abilities previously mentioned. Since the final store is not automatically transferred to the next time-unit, the process Pers represents the ability to remember all messages posted so far. Notice that processes Comp and Enc generate an infinite number of messages. E.g., if the current store is out\((m, m)\), the process Comp will add the constraints out\((\{m, m\}, m)\) and out\((\{m, \{m, m\}\})\) and so on.

To deal with this state explosion problem, the number of messages to be considered can be bound (see e.g. [29]). We formalize this with the following abstraction.

**Definition 18** (Abstract secure cons. system). Let \( M \) be the set of (terms) messages in the constraint system in Definition 17 and \( Ig \) be \( M \rightarrow \mathbb{N} \) defined as:

\[
Ig(m) = \begin{cases} 
0 & \text{if } m \in P \cup K \cup Var \\
1 + Ig(m_1) + Ig(k) & \text{if } m = [m_1]_k \\
1 + Ig(m_1) + Ig(m_2) & \text{if } m = [m_1, m_2] 
\end{cases}
\]

Let cut\(_{\alpha} \) be the following term abstraction

\[
cut\(_{\alpha}(m) = \begin{cases} 
m & \text{if } Ig(m) \leq \alpha \\
\text{otherwise} & \text{if } m \in M \setminus (representing all the messages with length greater than} \ \alpha \) \ . \ Let \ C \ be as in Definition 17 and \( (C, \alpha, A) \) be a description where
\[
\alpha(out(m)) = out(cut\(_{\alpha}(m)) \\
\alpha(secret(m)) = secret(cut\(_{\alpha}(m))
\]

### 5.3.1 Secret Key Analysis

Assume the following simplification of the Denning-Sacco key distribution protocol [9]:

\[
\begin{align*}
msg_1 & : A \rightarrow B : \{A, m\}_{pub(b)} \\
msg_2 & : B \rightarrow A : \{n\}_m
\end{align*}
\]

In the first message, A sends to B \( \{A, m\}_{pub(b)} \) representing the composition of the A’s identifier and the nonce (unguessable secret) \( m \) encrypted with the B’s public key. With its private key, B is able to decrypt the message sent by A and then creates the nonce \( n \). B sends \( n \) encrypted with \( m \). The goal of the protocol is to keep secret \( n \).

This protocol can be modeled in utcc as

\[
\begin{align*}
init(i, r) & \triangleright (local m)tell(out([i, m]_{pub(r)})) \\
resp(t) & \triangleright (abs p, y. out([p, y]_{pub(t)})) \\
& \quad (local n)tell(secret(n)) \\
& \quad next tell(out([n, y])) \\
& \quad next resp(t)
\end{align*}
\]

Nonce generation is modeled by local constructs and the process tell(out\((m)\)) models the broadcast of the message \( m \). Inputs (message reception) are modelled by abs processes. Both, init and resp are recursively called as principals may initiate different sessions during the execution of the protocol. Finally, tell(secret\((n)\)) in resp states that the nonce \( n \) cannot be revealed.

Assume an execution of the Denning-Sacco protocol with three principals \( A, B, C \) where \( A \) starts the protocol with \( B \) and the private key of \( B \) is known by the attacker:

\[
DS = Spy \parallel init(A, B) \parallel \prod_{c\in\{A,B,C\}} (resp(x)) \parallel \prod_{c\in\{A,B,C\}} (tell(out(pri(x))))
\]

The abstraction cut\(_{\alpha} \) and \( \tau = sequence(2) \) allows us to verify the following:

if \( s_\alpha \in [DS]^{\tau} \) then \( s_\alpha(2) \vdash^{\omega} \exists_\alpha(\text{secret}(n) \land \text{out}(n)) \)

meaning that \( DS \) leads to a secrecy attack. In fact, this is a well known attack (see e.g. [3]) where the attacker replies to \( C \) the message sent by \( A \) to \( B \) and \( C \) believes that he is establishing a session key with \( A \). Since the attacker knows \( n \) from the first message, she can decrypt \( [n]_{pub(m)} \) and \( n \) is not longer a secret between \( A \) and \( C \) as intended.

Notice the importance of having here a finite cut of the messages (terms) generated for the process Spy to compute \([DS]^\tau\). This allows us to restrict the set of terms considered by the abs operator and over-approximate its behavior.

### 5.3.2 A prototypical implementation

We have implemented our framework and the abstract domain for secrecy analysis in a prototype developed in Oz (http://www.mozart-oz.org/). This tool is described at http://www.lilx.polytechnique.fr/~colarte/prototype/ and allows the user to compute the least element of the abstract semantics of a process \( P \). The current implementation supports constraints as those used in the cryptographic constraint system (e.g., predicates of the form \( out(\text{enc}(x, pub(y))) \)). It implements the sequence\((\kappa)\) and cut\(_{\alpha} \) abstractions where \( \kappa \) and \( \kappa' \) are parameters specified by the user. We started by implementing the secrecy analysis since one of the most appealing application of the utcc calculus is the modeling and verification of security protocols. Our goal in the short term is to develop (or adapt from existing implementation) previously defined domains for logic programs such as those used in Section 5.1. This will then provide a valuable tool for the analysis of tcc and utcc programs.

The reader may find in the URL above a deeper description of the tool and some examples. Furthermore, we provide the program excerpts to compute the output of the secrecy analysis for the Denning-Sacco key distribution protocol [9]. We also illustrate a similar analysis for the Needham-Schroeder Protocol [19].

### 6. Concluding Remarks

Several frameworks and abstract domains for the analysis of logic programs have been defined (see e.g. [6, 5, 1]). Those wires differ from ours since they do not deal with the temporal behavior and synchronization mechanisms present in tcc-based languages. On the contrary, since our framework is parametric w.r.t the abstract domain, it can benefit from those works.

We defined in [13] a framework for the declarative debugging of ntc programs (a non-deterministic extension of tcc). The framework presented here is more general since it was designed for the static analysis of tcc and utcc programs and not only for debugging. Furthermore, as mentioned above, it is parametric w.r.t an abstract domain. The language utcc is also more involved: processes may exhibit infinite internal behavior and, unlike ntc, utcc can encode powerful formalisms [25]. In [13] we also dealt with infinite sequences of constraints and a similar finite cut over sequences was proposed there.

In [26] a symbolic semantics for utcc was proposed to deal with the infinite internal reductions of non well-terminated processes. This semantics, by means of temporal formulae, represents finitely the infinitely many constraints (and substitutions) the SOS may produce. The work in [25] introduces a denotational semantics for utcc based on (partial) closure operators over sequences of temporal logic formulæ. 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 and abstracted-unless free processes (see Definition 10). The semantics here presented turns out to be more appropriate than that in [25] to develop the abstract interpretation framework in Section 4. Firstly, the inclusion relation between the strongest postcondition and the semantics is verified for the whole language (The-
orem 3) — in [25] this inclusion is verified only for the abstracted—unless free fragment—. Secondly, this semantics makes use of the entailment relation over constraints rather than the more involved entailment over first-order linear-time temporal formulae as in [25]. This shall ease the implementation of tools based on the framework. Finally, our semantics allows us to capture the behavior of \textsc{UTCc} programs with recursion. This is not possible with the semantics in [25] which was thought only for \textsc{UTCc} programs where recursion can be encoded.

For the kind of applications that stimulated the development of \textsc{UTCc}, it was defined entirely deterministic. The semantics here presented could smoothly be extended to deal with some forms of non-determinism like those in [12], thus widening the spectrum of applications of our framework. It would be also interesting to study how our framework could adapt to stochastic and probabilistic extensions of csp-based languages which have found application e.g., in the modeling of biological systems [24].

In [26] the symbolic semantics and the underlying temporal logic associated to \textsc{UTCc} are used to verify a security protocol. The flaw in the protocol was exhibited by hand computing the symbolic outputs of the process. Here we go further by exhibiting the flaw automatically with the help of a prototype. Since our approach is based on approximations of the concrete semantics, not detecting a flaw does not imply the correctness of it.

As we showed in Section 5.1, given that \textsc{UTCc} is a sub-calculus of \textsc{UTCc}, our results apply straightforwardly to \textsc{UTCc} programs. This work then provides the theoretical basis for building tools for the data-flow analyses of \textsc{UTCc} and \textsc{UTCc} programs whose verification and debugging are not trivial due to their concurrent nature and synchronization mechanisms. We have shown for example, how to analyze groundness and how to detect mistakes in safety critical applications, such as control systems and embedded systems.

Our results should foster the development of analyzers for different concurrent systems modeled in \textsc{UTCc} and its sub-calculus such as security protocols, reactive and timed systems, biological systems, etc (see [24] for a survey of applications of csp-based languages). We plan also to perform freeness, suspension, type and independence analyses among others. It is well known that this kind of analyses have many applications, e.g. for code optimization in compilers, for improving run-time execution, and for approximated verification.

We believe that the framework proposed here can also help to develop new analyses for other languages for reactive systems (e.g. Esterel [2]), which can be translated into \textsc{UTCc} [30, 27] and for languages featuring mobile behavior as the \textsc{UTCc}—calculus [21]. For the latter, many analyses have been already defined, see e.g. [14, 15]. As future work, it would be interesting to see if it is possible to carry out similar analyses in our framework for suitable fragments of \(\pi\) that can be encoded into \textsc{UTCc} (see e.g., [18] that encodes a \(\pi\)-based language for structured communication into \textsc{UTCc}).

References


A. Proof Sketches

In this appendix we include the proof sketches for Theorems 3, 4 and 5. Before that, we require some additional results.

Proposition 1. Let \( P = (\text{abs } \bar{x}; c)Q \) be an abstracted-unless free process and \( s \) a sequence of constraints. For any \( t \in T^I \),\( s \models t \) if \( s \models P \) then \( s(t) \models c[\bar{t}][x] \) implies \( s \models sp(Q[\bar{t}][x]) \).

Proof. Let \( s = c_1,c_2,c_3\ldots \) and \( P = (\text{abs } \bar{x}; c)Q \). Assume \( s \models sp(P) \) and \( c_1 \models c[\bar{t}][x] \) for \( \bar{x} \neq \bar{t} \) (note that we can assume \( \bar{x} \neq \bar{t} \) since by alpha conversion, \( \bar{x} \) in \( P \) can be replaced by any \( \bar{y} \in \text{fv}(s) \cup \text{fv}(P) \)). Assume also that \( P = P_1 \overset{(c_1,c_2)}{\rightarrow} P_2 \overset{(c_2,c_3)}{\rightarrow} \ldots \)

Let \( Q_1 = Q_1[\bar{t}][x] \) and \( P_1 = P_1^I \). Since \( c_1 \models c[\bar{t}][x] \), by rule \( \text{R}_{\text{ABS}} \) in the operational semantics, there exists \( P_{1}^1,\ldots,P_{1}^m \) and \( Q_{1},\ldots,Q_{\bar{t}} \) s.t. \( (P_{1}^1 \| Q_1,\ldots Q_{\bar{t}}) \rightarrow^* (P_{1}^1 \| Q_1,\ldots Q_{\bar{t}}) \)\( \| Q_{\bar{t}} \rightarrow^* \). Since \( P_{1}^I \models P_1 \) and \( P_2 \models P_{1}^I \| Q_{\bar{t}} \), we have \( P_2 \models F(P_{1}^I \| Q_{\bar{t}}) \). where \( F \) is the future function in Table 1. Then, by the semantics of the parallel operator we have \( Q_1 \overset{(c_1,c_2)}{\rightarrow} Q_2 \overset{(c_2,c_3)}{\rightarrow} Q_3 \overset{(c_3,c_4)}{\rightarrow} \ldots \) and we conclude \( s \models sp(Q[\bar{t}][x]) \).

The only if part of the above proposition also holds. Nevertheless, we only require the if part for the proofs below.

Lemma 1. Let \( \llbracket \cdot \rrbracket_X \) as in Table 3. For any process \( P \), \( s_k \in \mathcal{A}^* \) and \( t \in T^I \), \( \llbracket t \rrbracket_X \neq \bar{x} \) and \( \bar{x} \notin \text{fv}(s_k) \) it holds: \( s_k \models [P[\bar{t}][x]]_X \iff s_k \in \llbracket (\text{local } \bar{x})(P \| \text{tell}(\bar{x} = t)) \rrbracket_X \).

Proof. Given two sequences \( s_1 \) and \( s_2 \), let \( s' = s_k \cup s_2 \cup s_2 \) denote the sequence where \( s(i) = s_1(i) \cup s_2(i) \) for \( i > 0 \). Furthermore, given a constraint \( c \in \mathcal{C} \), \( s \cup c \cdot c \cdot \ldots \). \( (\Rightarrow) \). Assume \( s_k \in [P[\bar{t}][x]]_X \) and \( \bar{x} \notin \text{fv}(s_k) \). Since \( \bar{x} \notin \text{fv}(P[\bar{t}][x]) \) we have \( s_k \cup \alpha(\bar{x} = \bar{t}) \models [P[\bar{t}][x]]_X \).

Given that \( \bar{x} \notin \text{fv}(s_k) \) then \( s_k \) is an \( \bar{x} \)-variant of \( s_k \). By Equation \( \text{D}_{\text{TELL}} \), \( s_k \in \llbracket (\text{tell}(\bar{x} = \bar{t})) \rrbracket_X \) and by Equations \( \text{D}_{\text{PAR}} \) and \( \text{D}_{\text{LOC}} \), \( s_k \in \llbracket (\text{local } \bar{x})(P[\bar{t}][x] \| \text{tell}(\bar{x} = t)) \rrbracket_X \). One can easily prove that \( [P[\bar{t}][x] \| \text{tell}(\bar{x} = t)]_X \models [P \| \text{tell}(\bar{x} = t)]_X \).

Therefore, \( s_k \in \llbracket (\text{local } \bar{x})(P \| \text{tell}(\bar{x} = t)) \rrbracket_X \). The proof of \( (\Rightarrow) \) can be obtained easily by reversing the previous arguments.

Theorem 2 (Completeness) Let \( D.P \) be a locally independent and abstracted-unless free program s.t. \( s \in [P]^I \). For all prefixes \( s' \) of \( s \), if there exists \( s'' \) s.t. \( s', s'' \) \( \in \text{io}(P) \) then \( s' \equiv s'' \). \( i.e., s' \notin sp(P) \).

Proof. The proof proceeds by induction on the structure of \( P \). We only outline the main steps of the case \( P = (\text{abs } \bar{x}; c)Q \). The other cases are rather standard in CCP. For extensiveness, if \( (s', s'') \in \text{io}(P) \) then \( s' < s'' \) \( i.e., s' \notin sp(P) \).

By inductive hypothesis, there is no a sequence \( \bar{w} \) s.t. \( s' \models c[\bar{t}][x] \) and \( s' \notin sp(Q[\bar{t}][x]) \). By inductive hypothesis, there is no a sequence \( \bar{w} \) s.t. \( s' \models c[\bar{t}][x] \). Since \( s' \models P \) and \( s' \models c[\bar{t}][x] \), by Equation \( \text{D}_{\text{ABS}} \) we have \( s \in [Q[\bar{t}][x]]_X \). Thus a contradiction.

Proposition 2. Let \( \llbracket \cdot \rrbracket_X \) as in Table 3 and \( \bar{t}_1,\bar{t}_2 \) be concrete terms different from \( \bar{x} \). \text{s.t.} \( [\bar{t}_1] = [\bar{t}_2] \) and \( \omega_{\bar{t}_1} = \omega_{\bar{t}_2} \).

Proof. Assume that \( s_{\bar{t}_2} \in \llbracket (\text{local } \bar{x})(P \| \text{tell}(\bar{x} = \bar{t}_1)) \rrbracket_X \). By Equations \( \text{D}_{\text{LOC}} \) and \( \text{D}_{\text{PAR}} \), \( s_{\bar{t}_2} \) is a variant of \( s_{\bar{t}_1} \). Since \( s_{\bar{t}_2} \in \llbracket P \rrbracket_X \) and \( s_{\bar{t}_1} \in \llbracket \text{tell}(\bar{x} = \bar{t}_2) \rrbracket_X \). By Property (4) in Definition 12, \( s_{\bar{t}_2} \in \llbracket \text{tell}(\bar{x} = \bar{t}_2) \rrbracket_X \). We conclude \( s \in \llbracket (\text{local } \bar{x})(P \| \text{tell}(\bar{x} = \bar{t}_2)) \rrbracket_X \).

Theorem 5 (Soundness of the approximation). Let \( A \) be upper correct w.r.t. \( C \). \((\mathcal{C}, \alpha, \mathcal{A})\) be a description and \( \tau \) be a sequence abstraction. Let \( \llbracket \cdot \rrbracket \) and \( \llbracket \cdot \rrbracket^+ \) be respectively as in Definitions 8 and 16. Given an utcc program \( D.P \), if \( s \in [P] \) then \( \tau(s) \in [P]^+ \).

Proof. The proof proceeds by induction on the structure of \( P \). We only describe the main steps that must be followed for each case. The cases of \( \text{skip} \) and \( \text{unless} \) are trivial since \( \tau(\mathcal{A}^*) \) approximates every possible concrete computation. The cases for \( \text{tell} \) and \( \text{local} \) are discarded making use of the properties of the abstraction \( \alpha \) (Definition 12). The cases of parallel composition, \( \text{next} \) and replication (\( \cdot \cdot \cdot \)) are easily discarded appealing to the inductive hypothesis on the subprocesses. The proof of \( \text{when} \) follows that in [12]. Finally, the case for the \( \text{abs} \) operator is discarded by using the Proposition 2 and the alternative view of the \( \text{abs} \) operator in terms of the parallel composition of \( \text{when} \) processes.

B. Encoding Recursion in UTCC

Assume a procedure definition \( p(x) : \cdot \) and let \( \text{call}_{p}(\cdot) \) be added to the constraint system. The encoding is defined as follows:

- For the process definitions: \( \tau(\text{g}(y)) : \cdot = \text{nil}(\text{abs } \bar{y}; \text{call}_{p}(\text{g}(y))) \hat{P} \) where \( \hat{P} \) is the process obtained by replacing in \( P \) any call \( p(x) \) by \( \text{tell}(\text{call}_{p}(x)) \).
- Analogously, the call \( p(x) \) in all its other occurrences is replaced by \( \text{tell}(\text{call}_{p}(x)) \).