A Framework for Abstract Interpretation of Timed Concurrent Constraint Programs

Moreno Falaschi, Carlos Olarte, Catuscia Palamidessi

A Framework for Abstract Interpretation of Timed Concurrent Constraint Programs

Moreno Falaschi Carlos Olarte Catuscia Palamidessi

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.

1 Introduction

Concurrent Constraint Programming (ccp) [29] 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., [29, 25, 27, 23, 28, 18].

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 ccp 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) [28], 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 [27] (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 [22]. Basically, the tcc ask operator **when** c **do** P is generalized by a parametric ask operator of the form (**abs** $\vec{x}; c)\, P$ called *abstraction*. Roughly speaking, an *ask* process of the form $P =$ **when** c **do** Q remains blocked until the store is strong enough to entail the constraint c and then P behaves as Q. In the case of $P = (\textbf{abs}\ \vec{x}; c)\, Q$, the process $Q[\vec{t}/\vec{x}]$ is executed for all term $\vec{t}$ such that the current store entails $c[\vec{t}/\vec{x}]$. Notice that when $\vec{x}$ is the empty vector, we recover the tcc ask operator.

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 operators over sequences of constraints in the lines of [28]. Our semantics is precise for tcc 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 [17], logical characterizations [21, 23, 27], semantics [28, 26, 23]).

The abstraction we propose proceeds in two-levels. First, we approximate the constraint system leading to an abstract constraint system. We give the sufficient conditions which have to be satisfied for ensuring the soundness of the abstraction. Next, 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 [25] for a survey of applications of `ccp`-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 `tcc` program. This analysis is proven useful to derive a property of a control system specified in `tcc`. The second scenario presents an abstraction of a cryptographic constraint system. We then use the abstract semantics to approximate the behavior of the protocol and exhibit a secrecy flaw in a security protocol programmed in `utcc`.

We have also developed a prototypical application of our framework and implemented the abstract domain for the verification of secrecy properties. The examples in Section 5.3 were automatically verified with this tool available at `http://www.lix.polytechnique.fr/~colarte/prototype/`. In this URL the reader can also find the complete outputs of these examples as well as the application of the framework for the verification of another protocol not described in this paper.

We believe that our results can also help to define analyses for other languages for modeling reactive systems, e.g. Esterel [2], and for mobile computations (e.g. for languages based on the π-calculus [22]). See the discussion on related work in Section 6.

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

framework are presented in Section 5. Section 6 concludes.

Due to a lack of space, the proofs are omitted; they are included in the extended version of this paper [13].

2 Preliminaries

ccp-based calculi are parametric in a *constraint system* specifying the basic constraints (e.g. $x \leq 42$) agents can tell and ask. Here we consider an abstract definition of such systems as lattices following [29]. The notion of constraint system as first-order formulae (e.g. in [27, 23]) 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

$$\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \mathtt{true}, \mathtt{false}, \mathit{Var}, \exists, d \rangle \text{ s.t.:}$$

- $\langle \mathcal{C}, \leq, \sqcup, \mathtt{true}, \mathtt{false} \rangle$ is a lattice with $\sqcup$ the *lub* operation (representing the logical *and*), and $\mathtt{true}$, $\mathtt{false}$ the least and the greatest elements in $\mathcal{C}$ respectively. Elements in $\mathcal{C}$ are called *constraints* with typical elements $c, c', d, d' ....$
- *Var* is a denumerable set of variables and for each $x \in \mathit{Var}$ the function $\exists_x : \mathcal{C} \to \mathcal{C}$ is a cylindrification operator satisfying: (1) $\exists_x c \leq c$. (2) If $c \leq d$ then $\exists_x c \leq \exists_x d$. (3) $\exists_x(c \sqcup \exists_x d) = \exists_x c \sqcup \exists_x d$. (4) $\exists_x \exists_y c = \exists_y \exists_x c$.
- For each $x, y \in \mathit{Var}$, $d_{xy} \in \mathcal{C}$ is a *diagonal element* and it satisfies: (1) $d_{xx} = \mathtt{true}$. (2) If z is different from x, y then $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$. (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 defining the hiding operator as we explain below. The diagonal elements are useful to model parameter passing in procedures calls. If $\mathbf{C}$ contains an equality theory, then d_{xy} can be thought as the formulae $x = y$.

We say that d *entails* c in $\mathbf{C}$ iff $c \leq d$ and we write $d \vdash c$. If $d \vdash c$ and $c \vdash d$ we write $d \equiv c$.

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

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

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 `ccp` model, the shared store of constraints grows monotonically, i.e., agents cannot drop information (constraints) from it. Then, a systems that changes the state of a signal (i.e., the value of a variable) cannot be modeled: The conjunction of the constraints $signal = on$ and $signal = off$ leads to an inconsistent store.

The timed `ccp` calculus (`tcc`) [28] extends `ccp` for reactive systems. Time is conceptually divided into *time intervals* (or *time units*). In a particular time interval, a `ccp` process P gets an input c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. The resting point determines also a residual process Q which is then executed in the next time unit. The resulting store d is not automatically transferred to the next time unit. This way, computations during a time-unit proceed monotonically but outputs of two different time-units are not supposed to be related to each other. Therefore, the variable *signal* 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 [23].

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

$$\begin{array}{rcl} P, Q & := & \mathbf{skip} \;\mid\; \mathbf{tell}(c) \;\mid\; \mathbf{when}\ c\ \mathbf{do}\ P \;\mid\; P \parallel Q \;\mid \\ & & (\mathbf{local}\,\vec{x}; c)\,P \;\mid\; \mathbf{next}\,P \;\mid\; \mathbf{unless}\ c\ \mathbf{next}\,P \;\mid \\ & & !\,P \;\mid\; p(\vec{x}) \end{array}$$

The process **skip** does nothing thus representing inaction. The process $\mathbf{tell}(c)$ adds c to the store in the current time interval making it available to the other processes. The *ask* process $\mathbf{when}\ c\ \mathbf{do}\ P$ 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 $P_{i_1} \parallel ... \parallel P_{i_n}$.

The process $(\mathbf{local}\,\vec{x}; c)\,P$ *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 $c = \mathtt{true}$, we shall simply write $(\mathbf{local}\,\vec{x})\,P$ instead of $(\mathbf{local}\,\vec{x}; \mathtt{true})\,P$.

We shall use $bv(Q)$ (resp. $fv(Q)$) to denote the set of *bound* (resp. *free*) variables occurring in Q.

$$
\begin{array}{l}
micCtrl(Error, Button) :- \\
\quad (\textbf{local}\, E', B', e, b)\, (\\
\quad\quad !\,\textbf{tell}(Error = [e \mid E'] \sqcup Button = [b|B']) \\
\quad\quad \parallel \textbf{when}\ \texttt{on} \sqcup \texttt{open}\ \textbf{do} \\
\quad\quad\quad !\,\textbf{tell}(e = yes \sqcup E' = [] \sqcup b = stop) \\
\quad\quad \parallel \textbf{when}\ \texttt{off}\ \textbf{do}\ (\textbf{tell}(e = no) \parallel \textbf{next}\, micCtrl(E', B')) \\
\quad\quad \parallel \textbf{when}\ \texttt{closed}\ \textbf{do}\ (\textbf{tell}(e = no) \parallel \textbf{next}\, micCtrl(E', B')))
\end{array}
$$

Figure 1: `tcc` model for a microwave controller.

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

Assume a (recursive) procedure definition $p(\vec{y}) :- P$ where $fv(P) \subseteq \vec{y}$. 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 **next** process to avoid non-terminating sequences of recursive calls during a time-unit (see [28, 23]).

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 this system.

2.2 Mobile behavior and UTCC

The `tcc` calculus lacks of mechanisms for name passing, i.e., mobility in the sense of the π-calculus [22]. Let us illustrate this with an example. Let $\texttt{out}(\cdot)$ be a constraint and let $P = \textbf{when}\ \texttt{out}(x)\ \textbf{do}\ R$ a system that must react when receiving a stimulus of the form $\texttt{out}(n)$ for $n > 0$. We notice that under input $\texttt{out}(42)$, P does not execute R since $\texttt{out}(42)$ does not entail $\texttt{out}(x)$ (i.e. $\texttt{out}(42) \not\vdash \texttt{out}(x)$). 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 $\texttt{out}(t)$ is entailed by the store.

In [27], `tcc` is extended for *mobile reactive* systems leading to *universal timed* `ccp` (`utcc`). To model mobile behavior, `utcc` replaces the `tcc` ask operation **when** c **do** P with a more general parametric ask construction, namely $(\textbf{abs}\ \vec{x}; c)\, P$. This process can be viewed as a λ-*abstraction* of the process P on the variables $\vec{x}$ under the constraint (or with the *guard*) c. Intuitively, $Q = (\textbf{abs}\ \vec{x}; c)\, P$ performs $P[\vec{t}/\vec{x}]$ in the current time interval for *all the terms* $\vec{t}$ s.t $c[\vec{t}/\vec{x}]$ is entailed by the current store. For example, $P = (\textbf{abs}\ x; \texttt{out}(x))\, R$ under input $\texttt{out}(42)$ executes $R[42/x]$.

From a programming point of view, we can then see the variables $\vec{x}$ in the abstraction $(\textbf{abs}\ \vec{x}; c)\, P$ 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* **when** c **do** P *with* $(\textbf{abs}\ \vec{x}; c)\, P$ *with the variables in* $\vec{x}$ *being pairwise distinct.*

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

The process $Q = (\textbf{abs}\ \vec{x}; c)\, P$ binds $\vec{x}$ in P and c. Therefore, we extend accordingly the sets $bv(Q)$ and $fv(Q)$ of bound and free variables. Furthermore Q evolves into **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) *Let* $\mathcal{D}$ *be a set of procedure declarations of the form* $p(\vec{y}) :\!- P$. *A* `utcc` *program takes the form* $\mathcal{D}.P$ *where* P *is a process. For every procedure name, we assume that there exists one and only one corresponding declaration in* $\mathcal{D}$.

Remark 1 *The* `utcc` *calculus was introduced in [27] 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. The reader can find in [13][Appendix B] the 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*

$$\begin{array}{lcl} P & = & (\textbf{local}\, x)\, (\textbf{tell}(\texttt{out}(x)) \parallel P') \\ Q & = & (\textbf{abs}\ z; \texttt{out}(z))\, Q' \end{array}$$

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

$$(\mathbf{local}\, x)\,(P' \parallel Q'[x/z])$$

where P' and Q' share the local variable x created by P. Then, any information produced by P' on x can be seen by Q' and vice versa.

2.3 Operational Semantics (SOS)

The structural operational semantics (SOS) of `tcc` and `utcc` considers *transitions* between process-store *configurations* $\langle P, c\rangle$ 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:*

1. *$P \equiv Q$ if they differ only by a renaming of bound variables (alpha-conversion).*
2. *$P \parallel \mathbf{skip} \equiv P$.*
3. *$P \parallel Q \equiv Q \parallel P$.*
4. *$P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$.*
5. *$P \parallel (\mathbf{local}\, \vec{x}; c)\, Q \equiv (\mathbf{local}\, \vec{x}; c)\,(P \parallel Q)$ if $\vec{x} \notin fv(P)$ (Scope Extrusion)*
6. *$(\mathbf{local}\, \vec{x}; c)\,(\mathbf{local}\, \vec{y}; d)\, P \equiv (\mathbf{local}\, \vec{x}; \vec{y}\,; c \wedge d)\, P$ if $\vec{x} \cap \vec{y} = \emptyset$ and $\vec{y} \notin fv(c)$.*

Extend $\equiv$ by decreeing that $\langle P, c\rangle \equiv \langle Q, c\rangle$ iff $P \equiv Q$.

Transitions are given by the relations $\longrightarrow$ and $\Longrightarrow$ in Table 1. The *internal* transition $\langle P, d\rangle \longrightarrow \langle P', d'\rangle$ should be read as "P with store d reduces, in one internal step, to P' with store d' ". The *observable transition* $P \stackrel{(c,d)}{\Longrightarrow} R$ 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 [23, 27] for further details. The rules are easily seen to realize the operational intuitions given above. As clarified below, the seemingly missing rules for **next** and **unless** processes are given by $\mathrm{R}_{\mathrm{OBS}}$.

Let $Q = (\mathbf{local}\, x; c)\, P$ in Rule $\mathrm{R}_{\mathrm{LOC}}$. 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 cylindrification operator $\exists_x$ in 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.

Internal Transitions :

$$\mathrm{R_{TELL}}\ \frac{}{\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d \sqcup c\rangle}$$

$$\mathrm{R_{PAR}}\ \frac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle}$$

$$\mathrm{R_{UNL}}\ \frac{d \vdash c}{\langle \mathbf{unless}\ c\ \mathbf{next}\ P, d\rangle \longrightarrow \langle \mathbf{skip}, d\rangle}$$

$$\mathrm{R_{LOC}}\ \frac{\langle P, c \sqcup (\exists \vec{x} d)\rangle \longrightarrow \langle P', c' \sqcup (\exists \vec{x} d)\rangle}{\langle (\mathbf{local}\, \vec{x}; c)\, P, d\rangle \longrightarrow \langle (\mathbf{local}\, \vec{x}; c')\, P', d \sqcup \exists \vec{x} c'\rangle}$$

$$\mathrm{R_{ABS}}\ \frac{d \vdash c[\vec{t}/\vec{x}] \qquad |\vec{t}| = |\vec{x}|}{\langle (\mathbf{abs}\ \vec{x}; c)\, P, d\rangle \longrightarrow \langle P[\vec{t}/\vec{x}] || (\mathbf{abs}\ \vec{x}; c \sqcup \vec{x} \neq \vec{t}\ )\, P, d\rangle}$$

$$\mathrm{R_{STR}}\ \frac{\gamma_1 \longrightarrow \gamma_2}{\gamma_1' \longrightarrow \gamma_2'}\ \text{if } \gamma_1 \equiv \gamma_1' \text{ and } \gamma_2 \equiv \gamma_2'$$

$$\mathrm{R_{CALL}}\ \frac{p(\vec{y}) :\!- P \in \mathcal{D}}{\langle p(\vec{x}), d\rangle \longrightarrow \left\langle \Delta^{\vec{x}}_{\vec{y}} P, d\right\rangle}$$

$$\mathrm{R_{REP}}\ \frac{}{\langle !\, P, d\rangle \longrightarrow \langle P || \mathbf{next}\, !\, P, d\rangle}$$

Observable Transition :

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

Table 1: Operational Semantics for `tcc` and `utcc`. $\equiv$ is given in Definition 4. In $\mathrm{R_{ABS}}$, $\vec{x} \neq \vec{t}$ denotes $\bigvee_{1 \leq i \leq |\vec{x}|} x_i \neq t_i$. If $|\vec{x}| = 0$, $\vec{x} \neq \vec{t}$ is defined as `false`.

Let $Q = (\mathbf{abs}\ \vec{x}; c)\, P$ in Rule $\mathrm{R_{ABS}}$. If the current store entails $c[\vec{t}/\vec{x}]$ then $P[\vec{t}/\vec{x}]$ is executed. Additionally, the abstraction persists in the current time interval to allow other potential replacements of $\vec{x}$ in P. Notice that the guard c is augmented with $\vec{x} \neq \vec{t}$ (syntactic difference) to avoid executing $P[\vec{t}/\vec{x}]$ again. We assume then the constraint "$\neq$" to be defined in the constraint system. Furthermore, without loss of generality (by alpha conversion), we assume that the variables in $\vec{x}$ does not occur in $\vec{t}$.

The rule $\mathrm{R_{CALL}}$ makes use of the diagonal elements (see Section 2) to model parameter passing as standardly done in `ccp` [29]. In this equation,

$$\Delta^{\vec{x}}_{\vec{y}} P = (\mathbf{local}\, \vec{a})\, (!\, \mathbf{tell}(d_{\vec{x}\vec{a}}) \parallel (\mathbf{local}\, \vec{y})\, (!\, \mathbf{tell}(d_{\vec{a}\vec{y}}) \parallel P))$$

where the variables in $\vec{a}$ are assumed to occur neither in the declaration nor in the process P, and $d_{\vec{x}\vec{y}}$ denotes the constraint $\bigsqcup_{1 \leq i \leq |\vec{x}|} d_{x_i y_i}$. Roughly speaking, $\Delta^{\vec{x}}_{\vec{y}}$ equates the actual parameters $\vec{x}$ and the formal parameters $\vec{y}$. What we observe is then the execution of $P[\vec{x}/\vec{y}]$.

Rule $\mathrm{R_{OBS}}$ says that an observable transition from P labeled with (c, d) is obtained from a terminating sequence of internal transitions from $\langle P, c\rangle$ to $\langle Q, d\rangle$. 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:*

$$\begin{aligned}\langle R, \mathtt{true}\rangle &\longrightarrow^* \langle (\mathbf{local}\, x; c)\,(P' \parallel (\mathbf{abs}\ z; \mathtt{out}(z))\, Q'), \exists_x(c)\rangle \\ &\longrightarrow^* \langle (\mathbf{local}\, x; c)\,(P' \parallel Q'[x/z] \parallel Q''), \exists_x(c)\rangle\end{aligned}$$

where $Q'' = (\mathbf{abs}\ z; \mathtt{out}(z) \sqcup x \not\doteq z)\, Q'$ and $c = \mathtt{out}(x)$. Notice that P' and $Q'[x/z]$ 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 (Behavior) *Let $s = c_1.c_2...c_i$ and $s' = c'_1.c'_2...c'_i$ be sequences of constraints. If $P = P_1 \xrightarrow{(c_1,c'_1)} P_2 \xrightarrow{(c_2,c'_2)} ...P_i \xrightarrow{(c_i,c'_i)}$, we write $P \xrightarrow{(s,s')}$. The set*

$$io(P) = \{(s, s') \mid P \xrightarrow{(s,s')}\}$$

denotes the input-output *behavior of P.*

In [27], the outputs of a `utcc` process were proven to be equivalent up to $\equiv$:

Theorem 1 (Determinism [27]) *Let P be a* `utcc` *process. If $P \xrightarrow{(s,s')}$ and $P \xrightarrow{(s,s'')}$ then $s' \equiv s''$*

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

We then define a monotonic process as follows:

Definition 6 (Monotonic Processes) *We say that P is a* monotonic process *iff 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 $io(P)$ is a *partial closure operator* [26], i.e., it is a function satisfying *extensiveness* and *idempotence*. Furthermore, if P is *monotonic*, $io(P)$ is a *closure operator* satisfying additionally monotonicity.

A pleasant property of closure operators is that they are uniquely determined by their set of fixpoints, 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 $\{s \mid (s, s) \in io(P)\}$.*

Intuitively, $s \in sp(P)$, iff P under input s cannot add any information whatsoever, i.e. s is a quiescent sequence for P. We 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 $io(\cdot)$ is a closure operator.

Theorem 2 *Let min be the minimum function w.r.t. the order induced by $\leq$. Given a monotonic* `utcc` *process P, $(s, s') \in io(P)$ 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 we develop in Section 4.

Our semantics is built on the closure operator semantics for `ccp` and `tcc` in [29, 28]. Unlike the denotational semantics for `utcc` in [26], 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 $T_{\mathcal{D}}$, which computes in a bottom-up fashion the *interpretation* of each procedure definition $p(\vec{x}) :- P$ in $\mathcal{D}$. Such an interpretation is given in terms of the set of the quiescent sequences for $p(\vec{x})$.

3.1 Compositional Semantics

Let *ProcHeads* denote the set of process names with their formal parameters and recall that $\mathcal{C}^\omega$ stands for the set of infinite sequences of constraints. We

shall call *Interpretations* the set of functions in the domain $ProcHeads \to \mathcal{P}(\mathcal{C}^\omega)$. The semantics is defined as a function $[\![\cdot]\!] : (ProcHeads \to \mathcal{P}(\mathcal{C}^\omega)) \to (Proc \to \mathcal{P}(\mathcal{C}^\omega))$ which given an interpretation I, 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 whatsoever. So, **skip** cannot add any information to any sequence (Equation $\mathrm{D}_{\mathrm{SKIP}}$). The sequences to which **tell**(c) cannot add information are those whose first element entails c (Equation $\mathrm{D}_{\mathrm{TELL}}$). A sequence is quiescent for $P \parallel Q$ if it is for P and Q (Equation $\mathrm{D}_{\mathrm{PAR}}$).

The process **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 $\mathrm{D}_{\mathrm{NEXT}}$). A similar explanation can be given for the process **unless** c **next** P (Equation $\mathrm{D}_{\mathrm{UNL}}$). A sequence s is quiescent for $!\,P$ if it is quiescent for every process of the form **next** $^n P$ with $n \geq 0$. Then, every suffix of s must be quiescent for P (Equation $\mathrm{D}_{\mathrm{REP}}$).

We say that s is an $\vec{x}$-variant of s' if $\exists_{\vec{x}} s(i) = \exists_{\vec{x}} s'(i)$ for $i > 0$ (i.e. s and s' differ only on the information about $\vec{x}$). A sequence s is quiescent for $Q = (\mathbf{local}\,\vec{x}; c)\,P$ if there exists an $\vec{x}$-variant s' of s s.t. s' is quiescent for P and $s'(1) \vdash c$. Hence, if P cannot add any information to s' then Q cannot add any information to s.

The abstraction process $(\mathbf{abs}\ x; c)\,P$ can be seen as the parallel composition $\prod_{\vec{t} \in \mathcal{T}^{|\vec{x}|}} (\mathbf{when}\ c\ \mathbf{do}\ P)[\vec{t}/\vec{x}]$ where $\mathcal{T}$ denotes the set of terms in the underlying constraint system. Therefore, the Equation $\mathrm{D}_{\mathrm{ABS}}$ is given in terms of the equation for the ask operator [28]: a sequence $d.s$ is quiescent for **when** c **do** P either if d does not entail c or if d entails c and $d.s$ is quiescent for P (Equation $\mathrm{D}_{\mathrm{ASK}}$). This way, s is quiescent for $(\mathbf{abs}\ x; c)\,P$, if for all term $\vec{t}$, $s(1) \vdash c[\vec{t}/\vec{x}]$ implies that s is quiescent for $P[\vec{t}/\vec{x}]$ (rule $\mathrm{D}_{\mathrm{ABS}}$).

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

The domain of the denotation is $\mathbb{E} = (E, \subseteq^c)$ where $E = \mathcal{P}(\mathcal{C}^\omega)$ and $\subseteq^c$ is a Smyth-like ordering defined as follows: Let $X, Y \in E$ and $\lesssim$ be the preorder s.t $X \lesssim Y$ iff for all $y \in Y$, there exists $x \in X$ s.t. $x \leq y$. $X \subseteq^c Y$ iff $X \lesssim Y$ and ($Y \lesssim X$ implies $Y \subseteq X$). The bottom of $\mathbb{E}$ is then $\mathcal{C}^\omega$ (the set of all the sequences). We do not consider the empty set to be part of the domain. Then, the top element is the singleton $\{\mathtt{false}^\omega\}$ (since `false` is the greatest element in $(\mathcal{C}, \leq)$).

Let us briefly elaborate on the choice of the domain above. The upward closure which is implicit in the Smyth powerdomain (in the sense that every set is equivalent to its upward closure) is necessary in order to deal correctly with the entailment of constraints ($d \vdash c$ iff $c \leq d$) and with the parallel operator (intersection). This would not be possible with the Hoare or with the Egli-Milner powerdomains, which are not upward closed. Roughly speaking, if we consider

for instance the Hoare powerdomain, then the fixpoint construction should start with a bottom defined as the interpretation which assigns to every process definition the empty set or the singleton $\{\mathtt{true}^\omega\}$. But in these interpretations the parallel composition of $\mathbf{tell}(c)$ with a call $p()$ would be empty, which does not correspond to the standard meaning for these operators. A similar situation arises when considering the Egli-Milner powerdomain.

Formally, the semantics is defined as follows:

Definition 8 (Concrete Semantics) *Let $[\![\cdot]\!]_I$ be defined as in Table 2. The semantics of a program $\mathcal{D}.P$ is defined as the least fixpoint of the continuous operator:*

$$T_{\mathcal{D}}(I)(p(\vec{y})) = [\![\Delta^{\vec{x}}_{\vec{y}} P']\!]_I \text{ if } p(\vec{x}) :- P' \in \mathcal{D}$$

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

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

Example 4 *Assume two constraints $\mathtt{out}_a(\cdot)$ and $\mathtt{out}_b(\cdot)$,intuitively representing outputs of names on two different channels a and b. Let $\mathcal{D}$ be the following procedure definitions*

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

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

$$\begin{array}{lll} I_0 & : & p \rightarrow \{c.c'.s \mid c \vdash \mathtt{out}_a(x) \text{ and } c' \vdash \mathtt{out}_a(y)\} \\ & & q \rightarrow \{c_1.s \mid c_1 \vdash \mathtt{out}_a(t) \text{ implies } c_1 \vdash \mathtt{out}_b(t)\} \\ & & r \rightarrow \mathcal{C}^\omega \cap \mathcal{C}^\omega = \mathcal{C}^\omega \\ I_1 & : & p \rightarrow I_0(p) \\ & & q \rightarrow \{c_1.c_2.s \mid c_i \vdash \mathtt{out}_a(t) \text{ implies } c_i \vdash \mathtt{out}_b(t)\ ,\ i\text{=}1,2\} \\ & & r \rightarrow I_0(p) \cap I_0(q) \\ \ldots & & \\ I_\omega & : & p \rightarrow I_0(p) \\ & & q \rightarrow \{s \mid (s(i) \vdash \mathtt{out}_a(t) \text{ imp. } s(i) \vdash \mathtt{out}_b(t) \text{ for } i > 0\} \\ & & r \rightarrow I_\omega(p) \cap I_\omega(q) \end{array}$$

where t denotes any term. In words, if $s \in [\![r()]\!]$ then $s(1) \vdash \mathtt{out}_a(x)$, $s(2) \vdash \mathtt{out}_a(y)$ and for $i \geq 1$, if $s(i) \vdash \mathtt{out}_a(t)$ then $s(i) \vdash \mathtt{out}_b(t)$

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

$P = (\mathbf{abs}\ x; c(x))\,\mathbf{tell}(c(x+1))$. Under input $c(1)$, this process will generate constraints of the form $c(2), c(3), ...$, thus never producing an observable transition (see [27] for details). 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 ϵ. Nevertheless, 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 *if and only if for every s such that $s(i) \neq$ `false` for each i, there exists s' such as $(s, s') \in io(P)$.*

The fragment of well-terminated processes is a meaningful one. For instance, it was shown to be enough to encode Turing-powerful formalisms in [26]. It has also found application, e.g., in multimedia interaction systems [24] and declarative interpretation of languages for structured communications [19].

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 $\mathcal{D}.P$, if $s \in sp(P)$ then there exists s' s.t. $s.s' \in [\![P]\!]$.*

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, 23] for details. Thus, similarly to [8, 23], completeness is verified 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 $\mathcal{D}.P$ by decreeing that P and all P_i in $p_i(\vec{x}) :- P_i \in \mathcal{D}$ satisfy the conditions above.*

Theorem 4 (Completeness) *Let $\mathcal{D}.P$ 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. $(s', s'') \in io(P)$ then $s' \equiv s''$, i.e., $s' \in sp(P)$.*

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`)

$\mathrm{D}_{\mathrm{SKIP}}$	$[\![\mathbf{skip}]\!]_I$	$=$	$\mathcal{C}^\omega$
$\mathrm{D}_{\mathrm{TELL}}$	$[\![\mathbf{tell}(c)]\!]_I$	$=$	$\{d.s \in \mathcal{C}^\omega \mid d \vdash c\}$
$\mathrm{D}_{\mathrm{PAR}}$	$[\![P \parallel Q]\!]_I$	$=$	$[\![P]\!]_I \cap [\![Q]\!]_I$
$\mathrm{D}_{\mathrm{NEXT}}$	$[\![\mathbf{next}\, P]\!]_I$	$=$	$\{d.s \in \mathcal{C}^\omega \mid s \in [\![P]\!]_I\}$
$\mathrm{D}_{\mathrm{UNL}}$	$[\![\mathbf{unless}\ c\ \mathbf{next}\, P]\!]_I$	$=$	$\{d.s \in \mathcal{C}^\omega \mid d \not\vdash c \text{ and } s \in [\![P]\!]_I\} \cup \{d.s \in \mathcal{C}^\omega \mid d \vdash c\}$
$\mathrm{D}_{\mathrm{REP}}$	$[\![!\, P]\!]_I$	$=$	$\{s \in \mathcal{C}^\omega \mid \text{ for all } s'', s' \text{ s.t. } s = s''.s',\ s' \in [\![P]\!]_I\}$
$\mathrm{D}_{\mathrm{LOC}}$	$[\![(\mathbf{local}\, \vec{x}; c)\, P]\!]_I$	$=$	$\{s \in \mathcal{C}^\omega \mid \text{ there exists an } \vec{x}\text{-variant } s' \text{ of } s \text{ s.t. } s'(1) \vdash c \text{ and } s' \in [\![P]\!]_I\}$
$\mathrm{D}_{\mathrm{ASK}}$	$[\![\mathbf{when}\ c\ \mathbf{do}\ P]\!]_I$	$=$	$\{d.s \in \mathcal{C}^\omega \mid d \vdash c \text{ and } d.s \in [\![P]\!]_I\} \cup \{d.s \in \mathcal{C}^\omega \mid d \not\vdash c\}$
$\mathrm{D}_{\mathrm{ABS}}$	$[\![(\mathbf{abs}\ \vec{x}; c)\, P]\!]_I$	$=$	$\bigcap_{\vec{t} \in \mathcal{T}^{\lvert\vec{x}\rvert}} [\![(\mathbf{when}\ c\ \mathbf{do}\ P)[\vec{t}/\vec{x}]]\!]_I$
$\mathrm{D}_{\mathrm{CALL}}$	$[\![p(\vec{x})]\!]_I$	$=$	$I(p(\vec{x}))$

Table 2: Semantic Equations for `tcc` and `utcc` constructs. In $\mathrm{D}_{\mathrm{ABS}}$, if $|\vec{x}| = 0$ then $\mathcal{T}^{|\vec{x}|}$ is defined as $\{\epsilon\}$

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 $\mathrm{D}_{\mathrm{ABS}}$. 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 [32].

Definition 11 *Given two constraint systems*

$$\begin{array}{lcl} \mathbf{C} & = & \langle \mathcal{C}, \leq \sqcup, \mathtt{true}, \mathtt{false}, \mathit{Var}, \exists, d \rangle \\ \mathbf{A} & = & \langle \mathcal{A}, \leq^\alpha \sqcup^\alpha, \mathtt{true}^\alpha, \mathtt{false}^\alpha, \mathit{Var}, \exists^\alpha, d^\alpha \rangle \end{array}$$

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

obvious way.

We shall use c_κ, d_κ to range over constraints in $\mathbf{A}$ and s_κ, s'_κ to range over sequences in $\mathcal{A}^\omega$ and $\mathcal{A}^*$. Let $\vdash^\alpha$ be defined as in the concrete counterpart, i.e. $c_\kappa \leq^\alpha d_\kappa$ iff $d_\kappa \vdash^\alpha c_\kappa$. The set of abstract terms is denoted by $\mathcal{T}_\kappa$ and ranged by $t_\kappa, t'_\kappa ...$

Following standard lines in [12, 32] we impose the following restrictions over α:

Definition 12 (Correctness) *Let $\alpha : \mathcal{C} \to \mathcal{A}$ be monotonic. We say that* $\mathbf{A}$ *is* upper correct *w.r.t the constraint system* $\mathbf{C}$ *if for all $c \in \mathcal{C}$ and $x, y \in \mathcal{V}$: (1) $\alpha(\exists_x c) = \exists^\alpha_x \alpha(c)$. (2) $\alpha(d_{xy}) = d^\alpha_{xy}$. (3) $\alpha(c \sqcup d) \vdash^\alpha \alpha(c) \sqcup^\alpha \alpha(d)$. Let $\alpha_t : \mathcal{T} \to \mathcal{T}_\kappa$ be the term-abstraction structurally based on α. Given the sequence of variables $\vec{x}$ and $\vec{t}$,$\vec{t'} \in \mathcal{T}^{|\vec{x}|}$, (4) $\alpha(c[\vec{t}/\vec{x}]) = \alpha(c[\vec{t'}/\vec{x}])$ whenever $\alpha_t(\vec{t}) = \alpha_t(\vec{t'})$.*

Conditions (1), (2) and (3) relate the cylindrification, diagonal and *lub* 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 informally says that substituting by 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) [29] be the concrete domain. In Hcs, a first-order language $\mathcal{L}$ with equality is assumed. The entailment relation is that one expects from equality, e.g., $[x|y] = [a|z]$ 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 iff$(x, [])$ meaning that x is a ground variable. Abstract terms are variables and the special term g meaning "ground". In this setting, $\alpha(x = [a])$ = iff$(x, [])$ (i.e., x is a ground variable). Furthermore $\alpha_t(a) = \alpha_t(b) = g$. By Condition (4) in Definition 12, $\alpha((x = [y])[a/y]) = \alpha((x = [y])[b/y])$ = iff$(x, [])$.*

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

Definition 13 (Approximations) *Let* $\mathbf{A}$ *be* upper correct *w.r.t* $\mathbf{C}$ *and $(\mathcal{C}, \alpha, \mathcal{A})$ be a description. Given $d_\kappa = \alpha(d)$, we say that d_κ is the best approximation of d. Furthermore, for all $c_\kappa \leq^\alpha d_\kappa$ we say that c_κ approximates d and we write $c_\kappa \propto d$. 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* $\tau : \mathcal{A}^\omega \cup \mathcal{A}^* \to \mathcal{A}^*$ *is a reductive* ($\tau(s_\kappa) \leq^\alpha s_\kappa$) *and monotonic operator. We lift* τ *to sets of sequences in the obvious way:* $\tau(S_\kappa) = \{s_\kappa \mid s_\kappa = \tau(s'_\kappa) \text{ and } s'_\kappa \in S\}$.

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

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

We require $\mathbb{A}$ to be noetherian (i.e., there are no infinite ascending chains). This guarantees that the fixpoint 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 $\mathrm{A_{ASK}}$, $\mathrm{A_{ABS}}$ and $\mathrm{A_{UNL}}$. The other cases are easier.

We notice that from the fact $\alpha(d) \vdash^\alpha \alpha(c)$ we cannot conclude $d \vdash c$. For example, let $d = (x = 1)$, $c = (x = 2)$ and $\mathit{iff}(\cdot)$ be as in Example 5. We have $\mathit{iff}(x, []) \vdash^\alpha \mathit{iff}(x, [])$ but $x = 1 \not\vdash x = 2$. Then, the equation $\mathrm{A_{ASK}}$ cannot be obtained from the equation $\mathrm{D_{ASK}}$ by simply replacing the condition $d \vdash c$ with $d_\kappa \vdash^\alpha \alpha(c)$. We thus follow [32, 11, 12] for the abstract semantics of the ask operator. Intuitively, the Equation $\mathrm{A_{ASK}}$ says that if the abstract computation proceeds, then every concrete computation it approximates proceeds too. This is formalized by the relation $d_\kappa \vdash_{\mathcal{A}} c$, meaning that the abstract constraint d_κ entails c if all concrete constraint approximated by d_κ entails c.

Definition 15 *Given* $d_\kappa \in \mathcal{A}$ *and* $c \in \mathcal{C}$, $d_\kappa \vdash_{\mathcal{A}} c$ *iff for all* $c' \in \mathcal{C}$ *s.t.* $d_\kappa \propto c'$, $c' \vdash c$.

In Equation $\mathrm{A_{ABS}}$, we compute the intersection over the abstract terms ($\mathcal{T}_\kappa$) and we replace $\vec{x}$ with a concrete term $\vec{t'}$ s.t. $\alpha_t(\vec{t'}) = \vec{t}_\kappa$. Notice that it may be the case that there exists $\vec{t_1}$, $\vec{t_2}$ s.t. $\alpha_t(\vec{t_1}) = \alpha_t(\vec{t_2}) = \vec{t}_\kappa$. Using property (4) in Definition 12, we can show that the choice of the concrete term is irrelevant (see [13][Appendix A]).

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

$$\begin{array}{rcl} [\![\mathbf{unless}\ c\ \mathbf{next}\ P]\!]^\tau_X & = & \tau(\{d_\kappa.s_\kappa \mid d_\kappa \not\vdash_{\mathcal{A}} c \text{ and } s_\kappa \in [\![P]\!]^\tau_X) \\ & & \cup\ \tau(\{d_\kappa.s_\kappa \mid d_\kappa \vdash_{\mathcal{A}} c) \end{array}$$

Nevertheless, this equation leads to a non safe approximation of the concrete semantics. This is because from $d_\kappa \not\vdash_{\mathcal{A}} c$ we cannot conclude that $d \not\vdash c$ where

$\alpha(d) = d_\kappa$. To see this, take $Q = \mathbf{unless}\ c\ \mathbf{next}\, P$ and d s.t. $d \vdash c$. Then $d.\mathtt{true}^\omega \in [\![Q]\!]$. Take c' s.t. $c' \not\vdash c$ and $c'_\kappa = \alpha(c') \leq^\alpha \alpha(d) = d_\kappa$. Then, $d_\kappa \propto c'$ and $d_\kappa \not\vdash_{\mathcal{A}} c$. If $P \neq \mathbf{skip}$, we have $d_\kappa.\mathtt{true}^* \notin [\![Q]\!]^\tau$.

Defining $d_\kappa \not\vdash_{\mathcal{A}} c$ as true iff $c' \not\vdash c$ for all c' approximated by d_κ does not solve the problem. This is because under this definition, $d_\kappa \not\vdash_{\mathcal{A}} c$ would not hold for any d_κ and c: `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 $Q = \mathbf{unless}\ c\ \mathbf{next}\, P$ than $\tau(\mathcal{A}^\omega)$, i.e. $[\![Q]\!]^\tau_X = [\![\mathbf{skip}]\!]^\tau_X$ (Rule $\mathrm{A_{UNL}}$).

We define formally the abstract semantics as follows:

Definition 16 *Let $[\![\cdot]\!]^\tau_X$ be as in Table 3. The abstract semantics of a program $\mathcal{D}.P$ is defined as the least fixpoint of the following continuous semantic operator:*

$$T^\alpha_{\mathcal{D}}(X)(p(\vec{x})) = [\![(\Delta^{\vec{y}}_{\vec{x}} P')]\!]^\tau_X \ \textit{if}\ p(\vec{y}) :\!- P' \in \mathcal{D}$$

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

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 [32, Proposition 3], we deduce the following:

$$\begin{array}{ll} \underline{\alpha}(E) & := \tau(\{\alpha(s) \mid s \in E\}) \\ \gamma(A) & := \{s \mid \tau(\alpha(s)) \in A\} \end{array}$$

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

$$\begin{array}{ll} \underline{\alpha}(I)(p) & := \tau(\{\alpha(s) \mid s \in I(p)\}) \\ \gamma(X)(p) & := \{s \mid \tau(\alpha(s)) \in X(p)\} \end{array}$$

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

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

$\mathrm{A}_{\mathrm{SKIP}}$	$[\![\mathbf{skip}]\!]^\tau_X$	$=$	$\tau(\mathcal{A}^\omega)$
$\mathrm{A}_{\mathrm{TELL}}$	$[\![\mathbf{tell}(c)]\!]^\tau_X$	$=$	$\tau(\{d_\kappa.s_\kappa \in \mathcal{A}^\omega \mid d_\kappa \vdash^\alpha \alpha(c)\})$
$\mathrm{A}_{\mathrm{PAR}}$	$[\![P \parallel Q]\!]^\tau_X$	$=$	$[\![P]\!]^\tau_X \cap [\![Q]\!]^\tau_X$
$\mathrm{A}_{\mathrm{NEXT}}$	$[\![\mathbf{next}\, P]\!]^\tau_X$	$=$	$\tau(\{d_\kappa.s_\kappa \in \mathcal{A}^* \mid s_\kappa \in [\![P]\!]^\tau_X\})$
$\mathrm{A}_{\mathrm{UNL}}$	$[\![\mathbf{unless}\ c\ \mathbf{next}\, P]\!]^\tau_X$	$=$	$\tau(\mathcal{A}^\omega)$
$\mathrm{A}_{\mathrm{REP}}$	$[\![!\, P]\!]^\tau_X$	$=$	$\tau(\{s_\kappa \in \mathcal{A}^* \mid$ for all s'_κ, w_κ s.t. $s_\kappa = w_\kappa.s'_\kappa$, $s'_\kappa \in [\![P]\!]^\tau_X\})$
$\mathrm{A}_{\mathrm{LOC}}$	$[\![(\mathbf{local}\, \vec{x}; c)\, P]\!]^\tau_X$	$=$	$\tau(\{s_\kappa \in \mathcal{A}^* \mid$ there exists a $\vec{x}$-variant s'_κ of s_κ s.t. $s'_\kappa(1) \vdash^\alpha \alpha(c)$ and $s'_\kappa \in [\![P]\!]^\tau_X\})$
$\mathrm{A}_{\mathrm{ASK}}$	$[\![\mathbf{when}\ c\ \mathbf{do}\ P]\!]^\tau_X$	$=$	$\tau(\{d_\kappa.s_\kappa \in \mathcal{A}^* \mid d_\kappa \not\vdash_{\mathcal{A}} c\}) \cup$ $\{d_\kappa.s_\kappa \in \mathcal{A}^* \mid d_\kappa \vdash_{\mathcal{A}} c$ and $d_\kappa.s_\kappa \in [\![P]\!]^\tau_X\}$
$\mathrm{A}_{\mathrm{ABS}}$	$[\![(\mathbf{abs}\ \vec{x}; c)\, P]\!]^\tau_X$	$=$	$\bigcap_{\vec{t_\kappa} \in \mathcal{T}_\kappa^{\lvert\vec{x}\rvert}} [\![(\mathbf{when}\ c\ \mathbf{do}\ P)[\vec{t'}/\vec{x}]]\!]^\tau_X$ where $\alpha_t(\vec{t'}) = \vec{t_\kappa}$
$\mathrm{A}_{\mathrm{CALL}}$	$[\![p(\vec{x})]\!]^\tau_X$	$=$	$X(p(\vec{x}))$

Table 3: Abstract denotational semantics for `utcc`. $\vdash_{\mathcal{A}}$ in Definition 15

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 `tcc` program. We then apply this analysis in the verification of a reactive system in `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 `utcc`.

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 `tcc` program. To this end, we shall use as concrete domain the Herbrand constraint system (Hcs) [29] (see Example 5).

Assume the following procedure definitions:

$$
\begin{array}{lcl}
gen_a(x) & = & (\mathbf{local}\, x')\,(!\,\mathbf{tell}(x = [a|x']) \parallel \\
 & & \quad \mathbf{when}\ go_a\ \mathbf{do\ next}\ gen_a(x') \parallel \\
 & & \quad \mathbf{when}\ stop_a\ \mathbf{do}\ !\,\mathbf{tell}(x' = [\,]))\\
 & & \\
gen_b(x) & = & (\mathbf{local}\, x')\,(!\,\mathbf{tell}(x = [b|x']) \parallel \\
 & & \quad \mathbf{when}\ go_b\ \mathbf{do\ next}\ gen_b(x') \parallel \\
 & & \quad \mathbf{when}\ stop_b\ \mathbf{do}\ !\,\mathbf{tell}(x' = [\,]))\\
 & & \\
append(x,y,z) & = & \mathbf{when}\ x = [\,]\ \mathbf{do}\ !\,\mathbf{tell}(y = z) \parallel \\
 & & \mathbf{when}\ \exists_{x',x''}(x = [x'\,|x'']\,)\ \mathbf{do} \\
 & & \quad (\mathbf{local}\, x'', z')\,(!\,\mathbf{tell}(x = [x'\mid x'']) \parallel \\
 & & \quad\quad\quad !\,\mathbf{tell}(z = [x'\mid z']) \parallel \\
 & & \quad\quad\quad \mathbf{next}\ append(x'', y, z'))
\end{array}
$$

The process $gen_a(x)$ adds to the stream x an "a" when the environment provides go_a as input. Under input $stop_a$, it terminates the stream binding its tail to the empty list. Let x_go_a and x_stop_a be two distinct variables different from x and x', and go_a and $stop_a$ be the constraints $x\_go_a = []$ and $x\_stop_a = []$. The process gen_b can be explained similarly. The process $append(x, y, z)$ binds z to the concatenation of x and y.

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

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

Notice that Pos does not distinguish between the empty list and a list of ground terms, i.e., $d_\kappa = \alpha_g(x = []) = \alpha_g(x = [a]) = \mathit{iff}(x, \{\})$. Therefore, we have $d_\kappa \not\vdash_{\mathcal{A}} x = []$ (see Definition 15). This means, e.g., that the semantics of $P = \mathbf{tell}(x = []) \parallel \mathbf{when}\ x = []\ \mathbf{do\ tell}(y = [])$ is (safely) approximated by $\mathit{iff}(x, [])$. Thus we lose the information added by $\mathbf{tell}(y = [])$.

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:

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

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

Let $\mathbf{A_g} = \langle \mathcal{A}, \leq^\alpha\ \sqcup^\alpha, \mathtt{true}^\alpha, \mathtt{false}^\alpha, Var, \exists^\alpha, d^\alpha \rangle$ be the abstract constraint system obtained from the reduced product ([6]) of the previous abstract domains. Elements $g, g' ... \in \mathcal{A}$ are tuples $\langle c_\kappa, d_\kappa \rangle$ where c_κ corresponds to groundness information and d_κ to type dependency information. The abstraction function is defined as expected, i.e., $\alpha(c) = g = \langle \alpha_g(c), \alpha_T(c) \rangle$. The operations $\sqcup^\alpha$, $\exists^\alpha$ correspond to logical conjunction and existential quantification over the components of the tuple. The diagonal element d_{xy} corresponds to $\langle x = y, x = y \rangle$. Finally, $\langle c_\kappa, d_\kappa \rangle \leq^\alpha \langle c'_\kappa, d'_\kappa \rangle$ if $c'_\kappa \Rightarrow c_\kappa$ and $d'_\kappa \Rightarrow d_\kappa$.

Let $\tau = sequence(\kappa)$ and $g_1.g_2....g_\kappa \in [\![Gen_a(x)]\!]^\tau$. By a derivation similar to that of Example 4, if there exists $i \in \{1,..,\kappa\}$ such that $g_i \vdash_{\mathcal{A}} stop_a$, one can show that there exists $\vec{x'} = x'_0, x'_1, ..., x'_i$ such that

$$g_i \vdash^\alpha \exists_{\vec{x'}} \left\langle \begin{array}{l} iff(x, x'_0) \sqcup \bigsqcup_{0 \le j < i} iff(x'_j, \{x'_{j+1}\}) \sqcup iff(x'_i, \{\}) \\ list(x, x'_0) \sqcup \bigsqcup_{0 \le j < i} list(x'_j, x'_{j+1}) \sqcup nil(x'_i) \end{array} \right\rangle ,$$

Thus, if $g_i \vdash_{\mathcal{A}} stop_a$ we can deduce that x is a list and x is a ground variable, i.e., $g_i \vdash^\alpha \langle iff(x, []), list(x)\rangle$.

Let $s_k = [\![Gen_a(x) \parallel Gen_b(y) \parallel append(x, y, z)]\!]^\tau$. If there exist $i, j \le \kappa$ s.t. $s_\kappa(i) \vdash_{\mathcal{A}} stop_a$ and $s_\kappa(j) \vdash_{\mathcal{A}} stop_b$, we can show that for $l \ge max(i, j)$, the variables x, y and z are list of ground elements. More precisely,

$$s_\kappa(l) \vdash^\alpha \langle iff(x, []) \sqcup iff(y, []) \sqcup iff(z, []), list(x) \sqcup list(y) \sqcup list(z)\rangle$$

5.2 Analysis of Reactive Systems

Synchronous data flow languages [2] such as Esterel and Lustre can be encoded as `tcc` processes [31, 28]. 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. More precisely, we shall use the groundness analysis above to verify if the simplified version of a control system for a microwave in Example 1 complies with its intended behavior: the door must be closed when it is turned on.

We assume `on`, `off`, `closed` and `open` be respectively the constraints $on = [], off = [], close = []$ and $open = []$ for variables $on, off, close$ and $open$ different from E and E'. The symbols yes and $stop$ denote constant symbols.

Our analysis consists in determining when the variable $Error$ is bound to a ground term. If the system is correct, it must happen when the the door is open and the microwave is turned on.

Let $\tau = sequence(\kappa)$ for a given κ. We can show that if

$$s_\kappa \in [\![micCtrl(Error, Button)]\!]^\tau$$

and $s_\kappa(i) \vdash_{\mathcal{A}} (\texttt{open} \sqcup \texttt{on})$, then $s_\kappa(i) \vdash^\alpha \langle iff(Error, []), list(Error)\rangle$, i.e., $Error$ is a ground variable.

We then conclude that the system effectively binds the list Error to a ground term whenever the system reaches an inconsistent state.

5.3 Analyzing Secrecy Properties

In [27] 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 non-well terminated process thus producing infinitely many internal reductions. In this section we show how a suitable abstraction of the cryptographic constraint system in [27] may allow us to bound the number

of messages to be considered in a secrecy analysis. Then we exhibit a well-known flaw in a security protocol.

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

Definition 17 *Let Σ be a signature with constant symbols in $\mathcal{P} \cup \mathcal{K}$, function symbols enc, pair, priv and pub and predicates* $\mathtt{out}(\cdot)$ *and* $\mathtt{secret}(\cdot)$*. Constraints in $\mathcal{C}$ are first-order formulae built over Σ.*

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

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

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

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 the processes *Comp* and *Enc* generate an infinite number of messages. E.g., if the current store is $\mathtt{out}(m)$, the process *Comp* will add the constraints $\mathtt{out}(\{m, m\})$, $\mathtt{out}(\{m, \{m, m\}\})$ and so on.

To deal with this state explosion problem, the number of messages to be considered can be bound (see e.g. [30]). We formalize this with the following abstraction.

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

$$
lg(m) = \begin{cases} 0 & \text{if} \quad m \in \mathcal{P} \cup \mathcal{K} \cup Var \\ 1 + lg(m_1) + lg(k) & \text{if} \quad m = \{m_1\}_k \\ 1 + lg(m_1) + lg(m_2) & \text{if} \quad m = \{m_1, m_2\} \end{cases}
$$

Let cut_κ be the following term abstraction

$$
cut_\kappa(m) = \begin{cases} m & \text{if} \ \ lg(m) \leq \kappa \\ m_\top & \text{otherwise} \end{cases}
$$

with $m_\top \notin \mathcal{M}$ (representing all the messages with length greater than κ) . Let $\mathcal{C}$ be as in Definition 17 and $(\mathcal{C}, \alpha, \mathcal{A})$ be a description where $\alpha(\mathtt{out}(m)) = \mathtt{out}(cut_\kappa(m))$ and $\alpha(\mathtt{secret}(m)) = \mathtt{secret}(cut_\kappa(m))$.

5.3.1 Secrecy Analysis

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

$$\begin{array}{lll} msg_1 & A \rightarrow B & : \ \{A, m\}_{pub(B)} \\ msg_2 & B \rightarrow A & : \ \{n\}_m \end{array}$$

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

$$\begin{array}{lll} \mathbf{init}(i, r) & :- & (\mathbf{local}\, m)\, \mathbf{tell}(\mathtt{out}(\{i, m\}_{pub(r)})) \\ & & \parallel \mathbf{next}\, \mathbf{init}(i, r) \\ \mathbf{resp}(t) & :- & (\mathbf{abs}\ p, y; \mathtt{out}(\{p, y\}_{pub(t)})) \\ & & \qquad (\mathbf{local}\, n)\, !\, \mathbf{tell}(\mathtt{secret}(n)) \parallel \\ & & \qquad\qquad \mathbf{next}\, \mathbf{tell}(\mathtt{out}(\{n\}_y)) \\ & & \parallel \mathbf{next}\, \mathbf{resp}(t) \end{array}$$

Nonce generation is modeled by **local** constructs and the process $\mathbf{tell}(\mathtt{out}(m))$ models the broadcast of the message m. Inputs (message reception) are modelled by **abs** processes. Both, **init** and **resp** are recursively called since principals may initiate different sessions during the execution of the protocol. Finally, $\mathbf{tell}(\mathtt{secret}(n))$ 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:

$$DS = Spy \parallel init(A, B) \parallel \prod_{x \in \{A,B,C\}} (resp(x)) \ \parallel \mathbf{tell}(\mathtt{out}(priv(B)))$$

The abstraction cut_3 and $\tau = sequence(2)$ allows us to verify the following:

$$\text{if } s_\kappa \in [\![DS]\!]^\tau \text{ then } s_\kappa(2) \vdash^\alpha \exists_n(\mathtt{secret}(n) \sqcup \mathtt{out}(n))$$

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 m 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 the behavior of the protocol.

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 in `http://www.lix.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 $\texttt{out}(enc(x, pub(y)))$). It implements the $sequence(\kappa)$ and $cut_{\kappa'}$ abstractions where κ and κ' 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 then will 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 [20].

6 Concluding Remarks

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

We defined in [14] a framework for the declarative debugging of `ntcc` [23] 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 `ntcc`, `utcc` can encode Turing powerful formalisms [26]. In [14] we also dealt with infinite sequences of constraints and a similar finite cut over sequences was proposed there.

In [27] 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 [26] introduces a denotational semantics for `utcc` based on (partial) closure operators over sequences of *temporal logic formulae*. 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 [26] to develop the abstract interpretation framework in Section 4. Firstly, the inclusion relation between the strongest postcondition and the semantics is verified for the whole language (Theorem 3)

– in [26] 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 [26]. This shall ease the implementation of tools based on the framework. Finally, our semantics allows us to capture the behavior of `tcc` programs with recursion. This is not possible with the semantics in [26] which was thought only for `utcc` programs where recursion can be encoded.

For the kind of applications that stimulated the development of `utcc`, it was defined entirely deterministic. The semantics here presented could smoothly be extended to deal with some forms of non-determinism like those in [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 `ccp`-based languages which have found application e.g., in the modeling of biological systems [25].

In [27] the symbolic semantics and the underlying temporal logic associated to `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 `tcc` is a sub-calculus of `utcc`, our results apply straightforwardly to `tcc` programs. This work then provides the theoretical basis for building tools for the data-flow analyses of `utcc` and `tcc` 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 systems modeled in `utcc` and its sub-calculi such as security protocols, reactive and timed systems, biological systems, etc (see [25] for a survey of applications of `ccp`-based languages). We plan also to perform freeness, suspension, type and independence analyses among others. It is well known that this kind of analyses have many applications, e.g. for code optimization in compilers, for improving run-time execution, and for approximated verification.

We 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 `tcc` [31, 28] and for languages featuring mobile behavior as the π-calculus [22]. For the latter, many analyses have been already defined, see e.g. [15, 16]. 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 π that can be encoded into `utcc` (see e.g., [19] that encodes a π-based language for structured communication into `utcc`).

References

[1] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Two classes of Boolean functions for dependency analysis. *Science of Computer Programming*, 31(1), 1998.

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

[3] B. Blanchet. Security protocols: from linear to classical logic by abstract interpretation. *Inf. Process. Lett.*, 95(5):473–479, 2005.

[4] M. Codish and B. Demoen. Deriving polymorphic type dependencies for logic programs using multiple incarnations of prop. In *Proc. of SAS'94*, pages 281–296. Springer-Verlag, LNCS 864, 1994.

[5] M. Codish, H. Søndergaard, and P. Stuckey. Sharing and groundness dependencies in logic programs. *ACM Trans. Program. Lang. Syst.*, 21(5), 1999.

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

[7] F. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Information and Computation*, 161(1):45–83, 2000.

[8] F. S. de Boer, A. D. Pierro, and C. Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theor. Comput. Sci.*, 151(1):37–78, 1995.

[9] D. Denning and G. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8), 1981.

[10] D. Dolev and A. C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(12), 1983.

[11] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Compositional analysis for concurrent constraint programming. In *Proc. of LICS'93*, 1993.

[12] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. *Theoretical Computer Science*, 183(2):281–315, 1997.

[13] M. Falaschi, C. Olarte, and C. Palamidessi. A framework for abstract interpretation of timed concurrent constraint programs (extended version), 2009. `http://www.lix.polytechnique.fr/~colarte/`.

[14] M. Falaschi, C. Olarte, C. Palamidessi, and F. Valencia. Declarative diagnosis of temporal concurrent constraint programs. In *Proc. of ICLP'07*. Springer LNCS 4670, 2007.

[15] J. Feret. Abstract interpretation of mobile systems. *J. Log. Algebr. Program.*, 63(1):59–130, 2005.

[16] P.-L. Garoche, M. Pantel, and X. Thiroux. Abstract interpretation-based static safety for actors. *Journal of Software*, 2(3):87–98, 2007.

[17] T. Hildebrandt and H. A. Lopez. Types for secure pattern matching with local knowledge in universal concurrent constraint programming. In *Proc. of ICLP'09*. Springer LNCS, 2009.

[18] R. Jagadeesan, W. Marrero, C. Pitcher, and V. A. Saraswat. Timed constraint programming: a declarative approach to usage control. In *Proc. of PPDP'05*. ACM, 2005.

[19] H. Lopez, C. Olarte, and J. A. Pérez. Towards a unified framework for declarative structured communications. In *Proc. of PLACES'09*, 2009.

[20] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Proc. of TACAS'96*. LNCS, 1996.

[21] N. P. Mendler, P. Panangaden, P. J. Scott, and R. A. G. Seely. A logical view of concurrent constraint programming. *Nord. J. Comput.*, 2(2):181–220, 1995.

[22] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[23] M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(1):145–188, 2002.

[24] C. Olarte and C. Rueda. A declarative language for dynamic multimedia interaction systems. In *Proc of. MCM'09*. Springer, 2009.

[25] C. Olarte, C. Rueda, and F. Valencia. Concurrent constraint programming: Calculi, languages and emerging applications. *Newsletter of the ALP*, 21(2-3), 2008.

[26] C. Olarte and F. Valencia. The expressivity of universal timed CCP: Undecidability of monadic FLTL and closure operators for security. In *Proc. of PPDP 08*. ACM, 2008.

[27] C. Olarte and F. Valencia. Universal concurrent constraint programming: Symbolic semantics and applications to security. In *Proc. of SAC'08*. ACM, 2008.

[28] V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of LICS'94*. IEEE Computer Society, 1994.

[29] V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundation of Concurrent Constraint Programming. In *POPL'91*. ACM, 1991.

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

[31] S. Tini. On the expressiveness of timed concurrent constraint programming. *Electr. Notes Theor. Comput. Sci.*, 27, 1999.

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