



# The Expressivity of Universal Timed CCP: Undecidability of Monadic FLTL and Closure Operators for Security

Carlos Olarte<sup>1</sup> and Frank D. Valencia<sup>2</sup>

<sup>1</sup> INRIA, LIX, École Polytechnique, France  
colarte@lix.polytechnique.fr

<sup>2</sup> CNRS, LIX, École Polytechnique, France  
fvalenci@lix.polytechnique.fr

**Abstract.** The timed concurrent constraint programming model (*tcc*) is a *declarative* framework, closely related to First-Order Linear Temporal Logic (FLTL), for modeling *reactive systems*. 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*.

This paper is devoted to the study of 1) the expressiveness of *utcc* and 2) its semantic foundations. As applications of this study, we also state 3) a noteworthy decidability result for the well-established framework of FLTL and 4) bring new semantic insights into the modeling of security protocols.

More precisely, we show that in contrast to *tcc*, *utcc* is Turing-powerful by encoding *Minsky machines*. The encoding uses a *monadic* constraint system allowing us to prove a new result for a fragment of FLTL: The undecidability of the validity problem for monadic FLTL without equality and function symbols. This result refutes a decidability conjecture for FLTL from a previous paper. It also justifies the restriction imposed in previous decidability results on the quantification of flexible-variables.

We shall also show that as in *tcc*, *utcc* processes can be semantically represented as partial closure operators. The representation is fully abstract wrt the input-output behavior of processes for a meaningful fragment of the *utcc*. This shows that mobility can be captured as closure operators over an underlying constraint system. As an application we identify a language for security protocols that can be represented as closure operators over a cryptographic constraint system.

**Keywords:** Concurrent Constraint Programming, First-order Linear Temporal Logic, Closure Operators, Security Protocols

## 1 Introduction

Timed concurrent constraint programming (*tcc*) [24] is a declarative temporal formalism from concurrency theory for modeling reactive systems. The *tcc* calculus combines the traditional operational view of process calculi with a declarative one based upon first-order linear-time temporal logic (FLTL). The *tcc*

language is parametric in an underlying constraint system specifying the basic constraints (pieces of information) that agents can tell or ask during execution (e.g,  $x > 42$ ).

Both the computational *expressiveness* and *semantic* foundations of **tcc** have been thoroughly studied in the literature [24, 22, 28]. This allowed for a better understanding of **tcc** and its relation with other formalisms. In particular, the expressiveness study in [28] shows that **tcc** processes can be represented as *finite-state* Büchi automata [8]. The *denotational semantics* study in [24] shows that **tcc** processes can be compositionally interpreted as *closure operators* (monotonic, idempotent, and extensive functions) over sequences of constraints.

Universal **tcc** [23], henceforth **utcc**, is an extension of the **tcc** calculus. The purpose of the extension is to add to **tcc** the expressiveness needed for modeling *mobility* while preserving its elegant closure-operator semantic characterization. (Mobility refers to the communication of private names as typically done for *mobile systems* and *security protocols* in Concurrency Theory). Although [23] illustrates several **utcc** examples involving mobility, no formal statement about the expressiveness of **utcc** nor the preservation of the closure-operator semantics has been previously given.

In this paper we shall study the expressiveness of **utcc** in terms of its computational power and semantic characterization. We shall show that 1) unlike **tcc** processes, **utcc** processes can represent Turing-powerful formalisms. Furthermore, we shall show that despite this extra expressiveness as **tcc** processes 2) **utcc** processes can still be compositionally interpreted as partial closure operators. We shall also show the applicability of this declarative framework and the elegant theory of concurrent constraint programming to other domains of computer science. By using 1) we shall state a new insightful undecidability result for the well-established framework of FLTL. As an application of 2) we shall also bring new semantic insights into the modeling of security protocols. These two applications correspond to 3) and 4) in the contributions below.

*Contributions.* More precisely, to show 1) above we shall encode in **utcc** Minsky machines [20]. The encoding of Minsky machines in **utcc** will be given in terms of a very simple decidable constraint system involving only monadic predicates and no equality nor function symbols. The importance of using such a constraint system is that it will allow us to use **utcc** to state a new impossibility result for monadic FLTL and the main technical application of this paper: Namely, 3) the undecidability of the validity problem for monadic FLTL without functions symbols and equality. In fact, we show that this FLTL is strongly incomplete which of course implies its undecidability. Notice that this result contrasts the standard decidability result of (untimed) monadic *classic* first-order logic [7].

It is worth noticing that there are several works in the literature addressing the decidability of fragments of FLTL and in particular the monadic one [1, 18, 27, 16, 21]. The above-mentioned result is insightful in that it answers an issue raised in a previous work and justifies some restrictions on monadic FLTL in other decidability results by showing them to be necessary. Namely, in [28] it was suggested that one could dispense with the restriction to negation-free formula

in the decidability result for the Pnueli's FLTL fragment there studied. Our undecidability result actually contradicts this conjecture since with negation that logic would correspond to the FLTL here presented. Furthermore, another work in [18] shows the decidability of monadic FLTL. This seemingly contradictory statement arises from the fact that unlike this work, [18] disallows, without a technical explanation, quantification over flexible variables. Our results, therefore, show that restriction to be necessary for decidability thus filling a gap on the decidability study of monadic FLTL. We shall elaborate more on these related results in Section 4.2.

As for 2) above, by building on the semantics of  $\text{tcc}$  given in [24] we show that the input-output behavior of any  $\text{utcc}$  process can be characterized as a partial closure-operator. Because of additional technical difficulties posed by  $\text{utcc}$ , the codomain of the closure-operators is more involved than that for  $\text{tcc}$ . Namely, we shall use sequences of future-free temporal formulae (constraints) rather than sequences of basic constraints as in  $\text{tcc}$ . We shall also give a compositional denotational account of this closure-operator characterization. We shall show that closure-operator denotation of  $\text{utcc}$  is fully abstract wrt the input-output behavior of processes for a significant fragment of  $\text{utcc}$ . This in particular shows that mobility in  $\text{utcc}$  can be elegantly represented as closure operators over some underlying constraint system.

As an application of the above-mentioned semantic result, we shall 4) identify a process language for security protocols that can be represented as closure operators. This language arises as a specialization of  $\text{utcc}$  with a particular cryptographic constraint systems. We argue that the interpretation of the behavior of protocols as closure operators is a natural one: E.g., suppose  $f$  is a closure-operator denoting a spy inferring information from the one he sees on the network. By extensiveness ( $f(w) \succeq w$ ), the spy produces new information from the one he obtains, by monotonicity ( $w \succeq v$  implies  $f(w) \succeq f(v)$ ) the more information the spy gets, the more he will infer, and finally by idempotence ( $f(f(w)) = f(w)$ ) the spy infers as much as possible from the info he gets. To our knowledge no closure operator denotational account has previously been given in the context of calculi for security protocols.

All in all, this paper gives an expressiveness account of a declarative framework in terms of computational power and semantic characterization, and importantly, with novel applications to other domains, namely FLTL and security protocols.

*Organization.* The paper is organized as follows. Section 2 gives some preliminaries and Section 3 recalls the notions and definitions of  $\text{utcc}$ . The expressiveness study and its application to FLTL is presented in Section 4. The closure-operator semantics and its application to security protocols is given in Section 5. Finally, we discuss the related work in Section 6.

## 2 Preliminaries

In this section we recall the notions of constraint systems and Linear-Time Temporal Logic (FLTL).

*Constraint Systems.* Concurrent Constraint Programming (CCP) calculi [26] are parametric in a *constraint system*. A constraint system can be represented as a pair  $(\Sigma, \Delta)$  where  $\Sigma$  is a signature of function and predicate symbols, and  $\Delta$  is a first-order theory over  $\Sigma$ . Given a constraint system  $(\Sigma, \Delta)$ , let  $\mathcal{L}$  be its underlying first-order language with variables  $x, y, \dots$ , and the standard logic symbols  $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall, \text{true}$  and **false**. The set of *constraints*  $\mathcal{C}$  with typical elements  $c, d, \dots$  is given by the set of formulae over  $\mathcal{L}$  modulo logical equivalence. We say that  $c$  *entails*  $d$  in  $\Delta$ , written  $c \vdash_{\Delta} d$ , iff  $(c \Rightarrow d) \in \Delta$  (i.e., iff  $c \Rightarrow d$  is true in all models of  $\Delta$ ). We shall omit “ $\Delta$ ” in  $\vdash_{\Delta}$  when no confusion arises. For operational reasons  $\vdash$  is often required to be decidable.

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

*Temporal Formulae and Constraints.* In the following sections we shall make use of notions from First-Order Linear-Time Temporal Logic (FLTL). Here we recall the syntax and semantics of this logic. See [17] for further details.

**Definition 1.** *Given a constraint system with a first-order language  $\mathcal{L}$ , the LTL formulae we use are given by the syntax:*

$$F, G, \dots := c \mid F \wedge G \mid \neg F \mid \exists x F \mid \odot F \mid \circ F \mid \square F.$$

where  $c$  is a constraint in  $\mathcal{L}$ , from now on also referred to as state formula.

The modalities  $\odot F, \circ F$  and  $\square F$  stand for resp., that  $F$  holds *previously*, *next* and *always*. We use  $\forall x F$  for  $\neg \exists x \neg F$ , and the *eventual* modality  $\diamond F$  as an abbreviation of  $\neg \square \neg F$ .

We presuppose the reader is familiar with the basic concepts of Model Theory. The non-logical symbols of  $\mathcal{L}$  are given meaning in an underlying  $\mathcal{L}$ -structure, or  $\mathcal{L}$ -model,  $\mathcal{M}(\mathcal{L}) = (\mathcal{I}, \mathcal{D})$ . (They are interpreted via  $\mathcal{I}$  as relations over a domain  $\mathcal{D}$  of the corresponding arity). A *state*  $s$  is a mapping assigning to each variable  $x$  in  $\mathcal{L}$  a value  $s[x]$  in  $\mathcal{D}$ . This interpretation is extended to  $\mathcal{L}$ -expressions in the usual way (e.g.  $s[f(x)] = \mathcal{I}(f)(s[x])$ ). We write  $s \models_{\mathcal{M}(\mathcal{L})} c$  iff  $c$  is true wrt  $s$  in  $\mathcal{M}(\mathcal{L})$ . The state  $s$  is an  $x$ -variant of  $s'$  iff  $s'[y] = s[y]$  for each  $y \neq x$ . We use  $\sigma$  to range over infinite sequences of states. We say that  $\sigma$  is an  $x$ -variant of  $\sigma'$  iff for each  $i \geq 0$ ,  $\sigma(i)$  (the  $i$ -th state in  $\sigma$ ) is an  $x$ -variant of  $\sigma'(i)$ .

Furthermore, the variables are partitioned into *rigid* and *flexible* variables. For the rigid variables, each  $\sigma$  must satisfy the *rigidity* condition: If  $x$  is rigid

then for all  $s, s'$  in  $\sigma$   $s[x] = s'[x]$ . If  $x$  is a flexible variable then different states in  $\sigma$  may assign different values to  $x$ .

**Definition 2.** We say that  $\sigma$  satisfies  $F$  in an  $\mathcal{L}$ -structure  $\mathcal{M}(\mathcal{L})$ , written  $\sigma \models_{\mathcal{M}(\mathcal{L})} F$ , iff  $\langle \sigma, 0 \rangle \models_{\mathcal{M}(\mathcal{L})} F$  where:

$$\begin{array}{ll}
\langle \sigma, i \rangle \models_{\mathcal{M}(\mathcal{L})} \mathbf{true} & \\
\langle \sigma, i \rangle \not\models_{\mathcal{M}(\mathcal{L})} \mathbf{false} & \\
\langle \sigma, i \rangle \models_{\mathcal{M}(\mathcal{L})} c & \text{iff } \sigma(i) \models_{\mathcal{M}(\mathcal{L})} c \\
\langle \sigma, i \rangle \models_{\mathcal{M}(\mathcal{L})} \neg F & \text{iff } \langle \sigma, i \rangle \not\models_{\mathcal{M}(\mathcal{L})} F \\
\langle \sigma, i \rangle \models_{\mathcal{M}(\mathcal{L})} F \wedge G & \text{iff } \langle \sigma, i \rangle \models_{\mathcal{M}(\mathcal{L})} F \text{ and } \langle \sigma, i \rangle \models_{\mathcal{M}(\mathcal{L})} G \\
\langle \sigma, i \rangle \models_{\mathcal{M}(\mathcal{L})} \odot F & \text{iff } i > 0 \text{ and } \langle \sigma, i-1 \rangle \models_{\mathcal{M}(\mathcal{L})} F \\
\langle \sigma, i \rangle \models_{\mathcal{M}(\mathcal{L})} \circ F & \text{iff } \langle \sigma, i+1 \rangle \models_{\mathcal{M}(\mathcal{L})} F \\
\langle \sigma, i \rangle \models_{\mathcal{M}(\mathcal{L})} \Box F & \text{iff for all } j \geq i, \langle \sigma, j \rangle \models_{\mathcal{M}(\mathcal{L})} F \\
\langle \sigma, i \rangle \models_{\mathcal{M}(\mathcal{L})} \exists x F & \text{iff for some } x\text{-variant } \sigma' \text{ of } \sigma, \langle \sigma', i \rangle \models_{\mathcal{M}(\mathcal{L})} F
\end{array}$$

We say that  $F$  is valid in  $\mathcal{M}(\mathcal{L})$  iff for all  $\sigma$ ,  $\sigma \models_{\mathcal{M}(\mathcal{L})} F$ .  $F$  is said to be valid if  $F$  is valid for every model  $\mathcal{M}(\mathcal{L})$ .

**Definition 3 (FLTL Theories).** Given a constraint system  $(\Delta, \Sigma)$  with first-order language  $\mathcal{L}$ , the FLTL theory induced by  $\Delta$ ,  $T(\Delta)$  is the set of FLTL sentences that are valid in all the  $\mathcal{L}$ -structures (or  $\mathcal{L}$ -models) of  $\Delta$ . We write  $F \vdash_{T(\Delta)} G$  iff  $(F \Rightarrow G) \in T(\Delta)$ . We omit “ $(\Delta)$ ” in  $\vdash_{T(\Delta)}$  when no confusion arises.

*Closure Operators.* Finally we recall the notion of (partial) *closure operator* that we shall use in Section 5. Given a partially ordered set  $(S, \leq)$ , a *partial closure operator* on  $S$  is a function  $f : S \rightarrow S$  with the following properties: For all  $x \in S$

1. **Extensiveness:**  $x \leq f(x)$ .
2. **Idempotence:**  $f(f(x)) = f(x)$ .

Furthermore,  $f$  is a *closure operator* if it also satisfies

- 3 **Monotonicity:** for all  $x, y \in S$ , if  $x \leq y$  then  $f(x) \leq f(y)$ .

A fundamental derived property of a closure operator is that it is uniquely determined by its set of fixed points.

### 3 Universal Timed CCP

In this section we first describe the temporal concurrent constraint (tcc) model [24] following the presentation in [22]. We then recall the universal tcc model (utcc) introduced in [23].

*Timed CCP.* The  $\text{tcc}$  calculus extends CCP for timed systems [24]. 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 interval. The resulting store  $d$  is not automatically transferred to the next time interval.

**Definition 4.** Processes  $P, Q, \dots$  in  $\text{tcc}$  are built from constraints in the underlying constraint system by the following syntax:

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

with the variables in  $\vec{x}$  being pairwise distinct.

The process  $\mathbf{skip}$  does nothing and  $\mathbf{tell}(c)$  adds  $c$  to the store in the current time interval. If in the current time interval  $c$  can eventually be derived from the store, the *ask* process  $\mathbf{when} \ c \ \mathbf{do} \ P$  evolves into  $P$  within the same time interval. Otherwise  $\mathbf{when} \ c \ \mathbf{do} \ P$  evolves into  $\mathbf{skip}$ .  $P \parallel Q$  denotes  $P$  and  $Q$  running in parallel during the current time interval and  $(\mathbf{local} \ \vec{x}; c) P$  binds  $\vec{x}$  in  $P$  by declaring them private to  $P$  under a constraint  $c$ . The *bound variables*  $bv(Q)$  (*free variables*  $fv(Q)$ ) are those with a bound (a not bound) occurrence in  $Q$ .

The *unit-delay*  $\mathbf{next} \ P$  executes  $P$  in the next time interval. The *time-out*  $\mathbf{unless} \ c \ \mathbf{next} \ P$  is also an 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  $\mathbf{next}^n P$  as short for  $\mathbf{next} \dots \mathbf{next} \ P$ , with  $\mathbf{next}$  repeated  $n$  times. Finally, the *replication*  $!P$  means  $P \parallel \mathbf{next} \ P \parallel \mathbf{next}^2 P \parallel \dots$ , i.e., unboundedly many copies of  $P$  but one at a time.

### 3.1 The Language of UTCC

In [23] the authors introduced the *Universal Timed CCP* calculus ( $\text{utcc}$ ) by extending the  $\text{tcc}$  language with an abstraction operator. This calculus allows for mobility behavior in the sense of the  $\pi$ -calculus [19], i.e. generation and communication of private channels or links. In this section we recall the notion of abstractions in  $\text{utcc}$  central to its expressivity. Later we present its operational semantics. See [23] for further details.

In  $\text{utcc}$ , the ask operation  $\mathbf{when} \ c \ \mathbf{do} \ P$  is replaced with a *parametric ask* of the form  $(\mathbf{abs} \ \vec{x}; c) P$ . This process can be viewed as an *abstraction* of the process  $P$  on the variables  $\vec{x}$  under the constraint (or with the *guard*)  $c$ . From a programming language perspective,  $\vec{x}$  in  $(\mathbf{local} \ \vec{x}; c) P$  can be viewed as the local variables of  $P$  while  $\vec{x}$  in  $(\mathbf{abs} \ \vec{x}; c) P$  as the formal parameters of  $P$ .

From a logic perspective abstractions have a pleasant duality with the local operator: The processes  $(\mathbf{local} \ \vec{x}; c) P$  and  $(\mathbf{abs} \ \vec{x}; c) P$  correspond, resp., to the *existential* and *universal* formulae  $\exists \vec{x}(c \wedge F_P)$  and  $\forall \vec{x}(c \Rightarrow F_P)$  where  $F_P$  corresponds to  $P$ . We shall elaborate more on this logical meaning of  $\text{utcc}$  processes in Section 4.2.

**Definition 5 (utcc processes).** *The utcc processes result from replacing in the syntax in Definition 4 the expression **when**  $c$  **do**  $P$  with  $(\mathbf{abs} \vec{x}; c) P$  with the variables in  $\vec{x}$  being pairwise distinct.*

Intuitively,  $Q = (\mathbf{abs} \vec{x}; c) P$  executes  $P[\vec{t}/\vec{x}]$  in the current time interval for all the sequences of terms  $\vec{t}$  s.t.  $c[\vec{t}/\vec{x}]$  is entailed by the store. The process  $Q$  binds the variables  $\vec{x}$  in  $P$  and  $c$ . Sets  $fv(\cdot)$  and  $bv(\cdot)$ , are extended accordingly. Furthermore  $Q$  evolves into **skip** after the end of the time unit, i.e. abstractions are not persistent when passing from one time-unit to the next one.

**Notation 1** We use **when**  $c$  **do**  $P$  for the empty abstraction  $(\mathbf{abs} \epsilon; c) P$ . We write  $(\mathbf{local} \vec{x}) P$  as a short for  $(\mathbf{local} \vec{x}; \mathbf{true}) P$ . The derived operator  $(\mathbf{wait} \vec{x}; c) \mathbf{do} P$  waits, possibly for several time units until for some  $\vec{t}$ ,  $c[\vec{t}/\vec{x}]$  holds. Then it executes  $P[\vec{t}/\vec{x}]$ . More precisely,

$$\begin{aligned} (\mathbf{wait} \vec{x}; c) \mathbf{do} P \stackrel{\text{def}}{=} & (\mathbf{local} \text{stop}, \text{go}) \mathbf{tell}(\text{out}'(\text{go})) \\ & \parallel \mathbf{!} \mathbf{unless} \text{out}'(\text{stop}) \mathbf{next} \mathbf{tell}(\text{out}'(\text{go})) \\ & \parallel \mathbf{!} (\mathbf{abs} \vec{x}; c \wedge \text{out}'(\text{go})) (P \parallel \mathbf{!} \mathbf{tell}(\text{out}'(\text{stop}))) \end{aligned}$$

where  $\text{stop}, \text{go} \notin fv(P)$ . We shall use **whenever**  $c$  **do**  $P$  as a short for  $(\mathbf{wait} \epsilon; c) \mathbf{do} P$ .

**Mobile links.** We conclude this section with a small example from [23] illustrating how mobility is obtained from the interplay between abstractions and local processes.

*Example 1 (Mobility).* Let  $\Sigma$  be a signature with the unary predicates  $\text{out}_1, \text{out}_2, \dots$  and a constant 0. Let  $\Delta$  be the set of axioms over  $\Sigma$  valid in first-order logic. Let

$$\begin{aligned} P &= (\mathbf{abs} y; \text{out}_1(y)) \mathbf{tell}(\text{out}_2(y)) \\ Q &= (\mathbf{local} z) (\mathbf{tell}(\text{out}_1(z)) \parallel \\ & \quad \mathbf{when} \text{out}_2(z) \mathbf{do} \mathbf{next} \mathbf{tell}(\text{out}_2(0))) \end{aligned}$$

Intuitively, if a link  $y$  is sent on channel  $\text{out}_1$ ,  $P$  forwards it on  $\text{out}_2$ . Now,  $Q$  sends its private link  $z$  on  $\text{out}_1$  and if it gets it back on  $\text{out}_2$  it outputs 0 on  $\text{out}_2$ .

### 3.2 An Operational Semantics for UTCC

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

**Definition 6.** Let  $\equiv$  be the smallest congruence satisfying: 0)  $P \equiv Q$  if they differ only by a renaming of bound variables 1)  $P \parallel \mathbf{skip} \equiv P$ , 2)  $P \parallel Q \equiv Q \parallel P$ , 3)  $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$ , 4)  $P \parallel (\mathbf{local} \vec{x}; c) Q \equiv (\mathbf{local} \vec{x}; c) (P \parallel Q)$  if  $\vec{x} \notin fv(P)$ , 5)  $(\mathbf{local} \vec{x}; c) \mathbf{skip} \equiv \mathbf{skip}$ . Extend  $\equiv$  by decreeing that  $\langle P, c \rangle \equiv \langle Q, c \rangle$  iff  $P \equiv Q$ .

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

$R_T$	$\frac{}{\langle \mathbf{tell}(c), d \rangle \longrightarrow \langle \mathbf{skip}, d \wedge c \rangle}$
$R_P$	$\frac{\langle P, c \rangle \longrightarrow \langle P', d \rangle}{\langle P \parallel Q, c \rangle \longrightarrow \langle P' \parallel Q, d \rangle}$
$R_L$	$\frac{\langle P, c \wedge (\exists \vec{x}d) \rangle \longrightarrow \langle P', c' \wedge (\exists \vec{x}d) \rangle}{\langle (\mathbf{local} \vec{x}; c) P, d \rangle \longrightarrow \langle (\mathbf{local} \vec{x}; c') P', d \wedge \exists \vec{x}c' \rangle}$
$R_U$	$\frac{d \vdash c}{\langle \mathbf{unless} \ c \ \mathbf{next} \ P, d \rangle \longrightarrow \langle \mathbf{skip}, d \rangle}$
$R_R$	$\frac{}{\langle ! P, d \rangle \longrightarrow \langle P \parallel \mathbf{next} \ ! P, d \rangle}$
$R_A$	$\frac{d \vdash c[\vec{t}/\vec{x}] \quad  \vec{t}  =  \vec{x} }{\langle (\mathbf{abs} \ \vec{x}; c) P, d \rangle \longrightarrow \langle P[\vec{t}/\vec{x}] \parallel (\mathbf{abs} \ \vec{x}; c \wedge \vec{x} \neq \vec{t}) P, d \rangle}$
$R_S$	$\frac{\gamma_1 \longrightarrow \gamma_2 \quad \text{if } \gamma_1 \equiv \gamma'_1 \text{ and } \gamma_2 \equiv \gamma'_2}{\gamma'_1 \longrightarrow \gamma'_2}$
$R_O$	$\frac{\langle P, c \rangle \longrightarrow^* \langle Q, d \rangle \not\rightarrow}{P \xrightarrow{(c,d)} F(Q)}$

**Table 1.** Internal and observable reductions.  $\equiv$  and  $F$  are given in Definitions 6 and 7. In  $R_A$ ,  $\vec{x} \neq \vec{t}$  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**.

We only describe some of the rules in Table 1 due to space restrictions. The other rules are standard, easily seen to realize the operational intuitions given above (see [22] for further details). As clarified below, the seemingly missing rules for “next” and “unless” processes are given by  $R_O$ .

We dwell a little upon the description of Rule  $R_L$  as it may seem somewhat complex. Consider  $Q = (\mathbf{local} \ x; c) P$  in Rule  $R_L$ . 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 existentially quantifying  $x$  in  $d$ . Similarly, from the external point of view, the observable information about  $x$  that the reduction of internal agent  $P$  may produce (i.e.,  $c'$ ) cannot be observed. Thus we hide it by

existentially quantifying  $x$  in  $c'$  before adding it to the global store. Additionally, we make  $c'$  the new private store of the evolution of the internal process.

Rule  $R_A$  describes the behavior of  $(\mathbf{abs} \vec{x}; c) P$ . If the 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  $P$  but  $c$  is augmented with  $x_i \neq t_i$  to avoid executing  $P[\vec{t}/\vec{x}]$  again.

Rule  $R_O$  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 a  $\langle Q, d \rangle$ . 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.

**Definition 7.** Let  $F$  be a partial function defined as:

$$F(P) = \begin{cases} \mathbf{skip} & \text{if } P = \mathbf{skip} \\ \mathbf{skip} & \text{if } P = (\mathbf{abs} \vec{x}; c) Q \\ F(P_1) \parallel F(P_2) & \text{if } P = P_1 \parallel P_2 \\ (\mathbf{local} \vec{x}) F(Q) & \text{if } P = (\mathbf{local} \vec{x}; c) Q \\ Q & \text{if } P = \mathbf{next} Q \\ Q & \text{if } P = \mathbf{unless} c \mathbf{next} Q \end{cases}$$

**Notation 2** We shall denote with  $\mathcal{C}^\omega$  the set of infinite sequences of constraints with typical elements  $\alpha, \alpha', \dots$ . Given  $c \in \mathcal{C}$ ,  $c^\omega$  represents the constraint  $c.c.c\dots$ . The  $i$ -th element in  $\alpha$  is denoted by  $\alpha(i)$ .

*Input-Output Behavior.* We will use the following input-output relations over **utcc** processes.

**Definition 8 (Input-Output Relations).** Given  $\alpha = c_1.c_2\dots$  and  $\alpha' = c'_1.c'_2\dots$ , we write  $P \xrightarrow{(\alpha, \alpha')}$  whenever  $P = P_1 \xrightarrow{(c_1, c'_1)} P_2 \xrightarrow{(c_2, c'_2)} \dots$ . The set

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

denotes the input-output behavior of  $P$ . If  $io(P) = io(Q)$  we say that  $P$  and  $Q$  are input-output equivalent and we write  $P \sim^{io} Q$ . Furthermore, we say that  $P$  eventually outputs  $c$ , written  $P \Downarrow_c$ , if  $P \xrightarrow{(\mathbf{true}^\omega, \alpha')}$  and  $\alpha'(i) \vdash c$  for some  $i > 0$ .

*Remark 1.* Let  $P$  be a process and  $\alpha, \alpha'$  be sequences of constraints s.t.  $(\alpha, \alpha') \in io(P)$ . Recall that computation in **tcc** during a time-unit progresses via the monotonic accumulation of constraints [24]. Then for all  $i > 0$ ,  $c'(i) \vdash c(i)$ . Recall also that the final store at the end of the time-unit is not automatically transferred to the next one. Therefore, it may be the case that  $c(i)' \not\vdash c(i-1)'$ . Constraints in  $\alpha$  are provided by the environment as input to the system and then, they are not supposed to be related to each other.

In [23], **utcc** is shown to be *deterministic* in the following sense:

**Theorem 1 (Determinism [23]).** *Let  $\alpha, \alpha'$  and  $\beta$  be sequences of constraints. If both  $(\beta, \alpha), (\beta, \alpha') \in io(P)$  then for all  $i > 0$ ,  $\vdash \alpha(i) \Leftrightarrow \alpha'(i)$ . I.e. the outputs of  $P$  are the same up to logical equivalence.*

*Remark 2.* The reader may have noticed that the abstraction operator may induce an infinite sequence of internal transitions within a time interval thus never producing an observable transition. One source of infinite internal behavior involves looping (recursion) in abstractions. E.g.

$$R = (\mathbf{abs} \ x; \mathbf{out}_1(x)) ((\mathbf{local} \ z) \mathbf{tell}(\mathbf{out}_1(z)))$$

with  $\mathbf{out}_1$  as in Example 1. A similar problem involves several abstractions producing mutual recursive behavior. Another source of infinite internal behavior involves the constraint system under consideration. Let

$$R = (\mathbf{abs} \ x; c) P.$$

If the current store  $d$  entails  $c[t/x]$  for infinitely many  $t$ 's, then  $R$  will have to execute  $P[t/x]$  for each such  $t$ .

We shall see that there are meaningful **utcc** processes which do not exhibit infinite internal behavior. We call them *well-terminated* processes. In fact, we shall prove in the next section that they are sufficient to show that **utcc** is Turing-powerful.

**Definition 9 (Well-termination).** *The process  $P$  is said to be well-terminated iff for every  $\alpha$ , there exists  $\alpha'$  such as  $(\alpha, \alpha') \in io(P)$ .*

Nevertheless, in Section 5 we shall consider also non well-terminated processes. For this purpose, we will recall in Section 5.1 the alternative symbolic operational semantics given in [23] which deals with the above-mentioned internal termination problems.

## 4 Expressiveness of UTCC

In this section we shall state the Turing expressiveness of **utcc** with monadic first-order logic as underlying constraint system. We shall also state our main application result, namely the undecidability of the validity problem for Monadic FLTL without equality nor function symbols.

### 4.1 Turing-Expressiveness of UTCC

We shall begin by showing that the sub-language of well-terminated **utcc** processes is Turing-powerful. We prove this by providing a **utcc** construction (or encoding) of two-counter machines, also called *Minsky machines*, which are known to be Turing-powerful [20]. This construction will be defined in terms of a very

simple decidable underlying constraint system: The *monadic fragment of first-order logic without function symbols nor equality*. This is a key property for our undecidability result for FLTL.

A two-counter Minsky machine  $M(v_0, v_1)$  is an imperative program consisting of a sequence of labeled instructions  $L_1; \dots; L_k$  which modify the values of two non-negative counters  $c_0$  and  $c_1$  initially set to  $v_0$  and  $v_1$  (resp.). The instructions, using counters  $c_n$  for  $n \in \{0, 1\}$ , are of three kinds:

$$\begin{aligned} L_i &: \text{halt} \\ L_i &: c_n := c_n + 1; \text{goto } L_j \\ L_i &: \text{if } c_n = 0 \text{ then goto } L_j \text{ else } c_n := c_n - 1; \text{goto } L_k \end{aligned}$$

The Minsky machine *starts* at  $L_s$  and *halts* if the control reaches the location of a **halt** instruction. Furthermore, the Minsky machine  $M(v_0, v_1)$  *computes* the value  $n$  if it halts with  $c_0 = n$ .

We presuppose a monadic constraint system including the uninterpreted predicates  $\text{out}$ ,  $\text{out}_n^m(\cdot)$  for  $n \in \{0, 1\}$  and  $m \in \{1, 2\}$  as in Example 1. Additionally, we presuppose the following propositional variables:  $\text{isz}_n$ ,  $\text{inc}_n$ ,  $\text{dec}_n$  for  $n \in \{0, 1\}$  and **halt**. A counter is defined as follows  $c_n =$

$$\begin{aligned} &1 \text{ !when } \text{isz}_n \text{ do} \\ &2 \quad \text{unless } \text{inc}_n \text{ next tell}(\text{isz}_n) \parallel \\ &3 \quad \text{when } \text{inc}_n \text{ do next (local } a \text{) (tell}(\text{out}_n^1(a)) \parallel \\ &3' \quad \quad \quad \text{!when } \text{out}_n^2(a) \text{ do tell}(\text{isz}_n)) \\ &4 \parallel \text{(abs } z; \text{out}_n^1(z)) \\ &5 \quad \text{whenever } \text{dec}_n \vee \text{inc}_n \text{ do} \\ &6 \quad \quad \text{when } \text{dec}_n \text{ do next tell}(\text{out}_n^2(z)) \parallel \\ &7 \quad \quad \text{when } \text{inc}_n \text{ do next (local } b \text{) (tell}(\text{out}_n^1(b)) \parallel \\ &7' \quad \quad \quad \text{!when } \text{out}_n^2(b) \text{ do tell}(\text{out}_n^1(z))) \end{aligned}$$

The counters  $c_0$  and  $c_1$  are obtained by replacing the sub-index  $n$  in  $\text{out}_n^m(\cdot)$ ,  $\text{isz}_n$ ,  $\text{inc}_n$  and  $\text{dec}_n$  by 0 and 1 respectively in the expression above. Intuitively,  $\text{isz}_n$  is used to test if the counter is zero (line 1) and  $\text{inc}_n$  and  $\text{dec}_n$  to trigger the action of increment and decrement the counter respectively. Each time an increment instruction is executed, a new local variable is created and sent through the public channel  $\text{out}_n^1(\cdot)$  (lines 3 and 7). Decrement operations (line 6) send back these local variables using the channel  $\text{out}_n^2(\cdot)$ . The processes in lines 3' and 7' when receiving the correspondent local variable on channel  $\text{out}_n^2(\cdot)$  move to the state immediately before the last increment instruction took place.

For the set of instruction  $L_1; \dots; L_n$  we assume a set of variables  $l_1, \dots, l_n$ . The  $i$ -th instruction is encoded as

$$\llbracket L_i \rrbracket = \text{!when } \text{out}(l_i) \text{ do ins}(L_i)$$

where

$$ins(L_i) = \begin{cases} \mathbf{tell}(\mathbf{halt}) & \text{if } L_i = \mathbf{halt} \\ \mathbf{tell}(inc_n) \parallel \mathbf{next tell}(\mathbf{out}(l_j)) & \text{if } L_i = c_n := c_n + 1; \mathbf{goto } L_j \\ \mathbf{when } isz_n \mathbf{ do next tell}(\mathbf{out}(l_j)) \parallel & \text{if } L_i = \mathbf{if } c_n = 0 \mathbf{ then goto } L_j \\ \mathbf{unless } isz_n \mathbf{ next (tell}(dec_n) \parallel & \text{else } c_n := c_n - 1; \mathbf{goto } L_k \\ \mathbf{next tell}(\mathbf{out}(l_k))) & \end{cases}$$

Without loss of generality assume that counters are initially set to 0. The program takes the form:

$$\llbracket M(0, 0) \rrbracket = (\mathbf{local } l_1, \dots, l_n) (\mathbf{tell}(\mathbf{out}(l_s)) \parallel c_0 \parallel c_1 \parallel \mathbf{tell}(isz_0) \parallel \mathbf{tell}(isz_1) \parallel \llbracket L_1 \rrbracket \parallel \dots \parallel \llbracket L_n \rrbracket)$$

Let  $Dec_n$  be the process decrementing  $n$  times the counter  $c_0$ . If it succeeds, it outputs the constraint *yes*:

$$\begin{aligned} Dec_0 &= \mathbf{when } isz_0 \mathbf{ do tell}(yes) \\ Dec_n &= \mathbf{unless } isz_0 \mathbf{ next (tell}(dec_0) \parallel Dec_{n-1}) \end{aligned}$$

For the correctness of the construction, one can verify that  $M(0, 0)$  computes the value  $n$  if and only if after the encoding halts one can decrement  $c_0$  exactly  $n$  times until telling “yes”. More precisely:

**Theorem 2 (Correctness).** *A Minsky machine  $M(0, 0)$  computes the value  $n$  iff  $(\llbracket M(0, 0) \rrbracket \parallel \mathbf{whenever } \mathbf{halt } \mathbf{ do } Dec_n) \Downarrow_{yes}$*

As an application of the above construction we can show the undecidability of the input-output equivalence for well-terminated processes.

Given  $M(0, 0)$  we define  $\llbracket M(0, 0) \rrbracket'$  as the above encoding  $\llbracket M(0, 0) \rrbracket$  except that  $ins(\mathbf{halt}) = \mathbf{skip}$ —notice that  $ins(\mathbf{halt}) = \mathbf{tell}(\mathbf{halt})$  in  $\llbracket M(0, 0) \rrbracket$ . Clearly,  $M(0, 0)$  does not halt iff

$$\llbracket M(0, 0) \rrbracket' \sim^{io} \llbracket M(0, 0) \rrbracket.$$

We then obtain the following corollary.

**Corollary 1.** *Fix the underlying constraint system to be monadic first-order logic without equality nor function symbols. Then, the question of whether  $P \sim^{io} Q$ , given two well-terminated processes  $P$  and  $Q$ , is undecidable.*

A more compelling application of the above construction is given in the next section. In fact, the following result is the main application of the paper.

## 4.2 Application: Undecidability of monadic FLTL

In this section we shall state a new undecidability result for monadic FLTL as an application of the above Minsky encoding and the logic characterization of utcc in [23].

*FLTL Characterization.* We first need to recall the FLTL characterization of `utcc` [23].

**Definition 10.** Let  $\llbracket \cdot \rrbracket$  a map from `utcc` processes to FLTL formulae given by:

$$\begin{array}{lll} \llbracket \text{skip} \rrbracket & = \text{true} & \llbracket \text{tell}(c) \rrbracket = c \\ \llbracket (\text{abs } \vec{y}; c) P \rrbracket & = \forall \vec{y}(c \Rightarrow \llbracket P \rrbracket) & \llbracket P \parallel Q \rrbracket = \llbracket P \rrbracket \wedge \llbracket Q \rrbracket \\ \llbracket (\text{local } \vec{x}; c) P \rrbracket & = \exists \vec{x}(c \wedge \llbracket P \rrbracket) & \llbracket \text{next } P \rrbracket = \circ \llbracket P \rrbracket \\ \llbracket \text{unless } c \text{ next } P \rrbracket & = c \vee \circ \llbracket P \rrbracket & \llbracket ! P \rrbracket = \square \llbracket P \rrbracket \end{array}$$

The *compositional* encoding in Definition 10 gives a pleasant correspondence between `utcc` processes and FLTL formulae.

**Theorem 3 (Logic Correspondence [23]).** Let  $\llbracket \cdot \rrbracket$  as in Definition 10 and  $\vdash_T$  as in Definition 3. If  $P$  is a well-terminated process and  $c$  a state formula then

$$\llbracket P \rrbracket \vdash_T \diamond c \text{ iff } P \Downarrow_c .$$

It should be noticed that the encoding of Minsky machines in the previous section uses only well-terminated processes.

*Quantification of flexible variables.* In [18] a FLTL named TLV is studied. The logic we present in Definition 1 differs from TLV only in that TLV disallows quantification of flexible variables as well as the past operator. We shall see that quantification over flexible variables is fundamental for our encoding of Minsky machines. We also state in Corollary 2 that the past-free monadic fragment of the FLTL in Section 2 without equality nor function symbols is strongly incomplete. This in contrast with the same TLV fragment which is decidable wrt validity [18].

Because of the above-mentioned difference with TLV we shall use the following notation:

**Notation 3** Henceforth we use *TLV-flex* to denote the past-free fragment of the FLTL presented in Section 2, i.e., the set of FLTL formulae without occurrences of the past modality  $\ominus$ .

In [18] it is proven that the problem of validity of a monadic TLV formula  $A$  without equality and function symbols is decidable. This decidability result is proven the same way as the standard decidability result for classical monadic first-order logic—By getting rid of quantifiers and reducing the problem to the decidability of propositional LTL.

Nevertheless, quantification of flexible variables makes impossible to obtain a prenex form of a formula which may then allows us to get rid of quantifiers as it can be done for monadic first-order logic. Consider for example the unary predicate  $p_c(x)$  testing if  $x$  is equal to a constant  $c$ . Take the formula  $F = (p_c(x) \wedge \circ \neg p_c(x))$ . If  $x$  is a flexible variable, notice that  $\square \exists x F$  is satisfiable whereas  $\exists x \square F$  is not. Hence, moving quantifier to the outermost position to

get a prenex form does not preserve satisfiability. Notice also that if  $x$  is a rigid variable instead,  $\Box\exists xF$  and  $\exists x\Box F$  are both logically equivalent to **false**.

The following proposition follows from composing the encodings from Minsky machines into **utcc** and from **utcc** into FLTL.

**Proposition 1.** *Let  $M(0,0)$  be a Minsky machine and  $P = \llbracket M(0,0) \rrbracket$  as defined in Section 4.1. Let  $A = \llbracket P \rrbracket$  be the FLTL formula obtained as in Definition 10. Then  $A$  is a monadic without equality and function symbols TLV-**flex** formula.*

To see the importance of quantifying over flexible variables in the encoding  $A = \llbracket P \rrbracket$ , take the output of  $a$  in line 3 of the encoding  $P = \llbracket M(0,0) \rrbracket$  and assume that  $a$  is rigid. Notice that abstraction in line 4 is replicated, thus corresponding to a formula of the form  $\Box\forall z \text{out}_n^1(z) \Rightarrow F$ . Once the formula  $\text{out}_n^1(a)$  is true, by the rigidity of  $a$ , the formula  $F[a/z]$  must be true in the following states, which does not correspond to the intended meaning of the machine execution. Instead, if  $a$  is flexible, the fact that  $\text{out}_n^1(a)$  is true at certain state does not imply that  $F[a/z]$  must be true in the subsequent states.

Now with the help of the above proposition and the encoding we obtain the following:

**Proposition 2.** *Given a Minsky machine  $M(0,0)$ , it is possible to construct a monadic TLV-**flex** formula without equality and function symbols  $F_M$  s.t  $F_M$  is valid iff  $M(0,0)$  loops (i.e., it never halts).*

Let  $M(0,0)$  be a Minsky machine. Let  $P = \llbracket M(0,0) \rrbracket''$  where  $\llbracket M(0,0) \rrbracket''$  is defined as the encoding  $\llbracket M(0,0) \rrbracket$  in Section 4.1 except that  $\text{ins}(\cdot)$  adds **tell(running)** in parallel to the encoding of all instructions but **halt**. For the above proposition, we can then take  $F_M = A \Rightarrow \Box\text{running}$ , where  $A = \llbracket P \rrbracket$  and  $\llbracket P \rrbracket$  as given in Definition 10.

Since the set of looping Minsky machines (i.e. the complement of the halting problem) is not recursively enumerable, a finitistic axiomatization of monadic TLV-**flex** without equality and function symbols would yield to a recursively enumerable set of tautologies. Therefore such a logic is *strongly incomplete*.

**Corollary 2 (Incompleteness).** *Monadic TLV-**flex** without equality and function symbols is strongly incomplete.*

From this corollary it follows that the *validity problem* in the above-mentioned monadic fragment of TLV-**flex** is undecidable.

## 5 Closure-Operator Semantics

In the previous section we showed that **utcc** processes can represent Minsky machines. In this section we show that **utcc** processes can be represented as partial closure operators.

More precisely, we shall give a compositional characterization of the input-output behavior of **utcc** processes in terms of (partial) closure operators. We

build on the closure operator semantics in [24] for **tcc**. Because of additional technical difficulties posed by the abstraction operator of **utcc**, the codomain of the closure-operators is more involved than that for **tcc**. Namely, instead of sequences of constraints as in **tcc**, we shall use sequences of future-free temporal formulae (constraints).

The proposed denotational semantics takes into account also non well-terminated processes (Definition 9). Consequently, we shall use the alternative symbolic reduction semantics in [23] which deals with the infinite internal computation problems discussed in Section 3.2 (Remark 2).

## 5.1 Symbolic Semantics

Here we recall the symbolic reduction semantics in [23]. Intuitively, the symbolic observable reductions use temporal constraints to represent in a finite way a possibly infinite number of substitutions as well as the information that an infinite loop may provide. This semantics guarantees that every sequence of internal transitions is finite.

Before defining the symbolic semantics let us give some intuitions of its basic principles.

**A. Substitutions as Constraints.** Take  $R = (\mathbf{abs} \ x; c) P$ . The operational semantics performs  $P[t/x]$  for every  $t$  s.t  $c[t/x]$  is entailed by the store  $d$ . Instead, the symbolic semantics dictates that  $R$  should produce  $e = d \wedge \forall x (c \Rightarrow d')$  where, similarly,  $d'$  should be produced according to the symbolic semantics by  $P$ . Let  $t$  be an arbitrary term s.t  $d \vdash c[t/x]$ . The idea is that if  $e'$  is operationally produced by  $P[t/x]$  then  $e'$  should be entailed by  $d'[t/x]$ . Since  $d \vdash c[t/x]$  then  $e \vdash d'[t/x] \vdash e'$ . Therefore  $e$  entails the constraint that any arbitrary  $P[t/x]$  produces.

**B. Timed Dependencies in Substitutions.** The symbolic semantics represents as temporal constraints dependencies between substitutions from one time interval to another. E.g., suppose that for  $R$  above,  $P = \mathbf{next} \ \mathbf{tell}(c')$ . Operationally, once we move to the next time unit, the constraints produced are of the form  $c'[t/x]$  for those  $t$ 's s.t the final store  $d$  in the *previous* time unit entails  $c[t/x]$ . The symbolic semantics captures this as  $e' = (\ominus d) \wedge \forall x ((\ominus c) \Rightarrow c')$  where  $\ominus$  is the “previous” (or “past”) modality in FLTL (see Section 2).

For the symbolic semantics we then use the Future-free fragment of the FLTL in Definition 1.

**Definition 11 (Future-free constraints).** *A temporal formula is said to be future-free iff it does not contain occurrences of  $\square$  and  $\circ$ . We shall use  $e, e' \dots$  to range over future-free formulae and  $w, w', v, v', \dots$  to range over infinite sequences of future-free formulae. Notice that every constraint (i.e., state formula) is a future-free formula.*

We shall assume that processes and configurations are extended to include future-free formulae rather than just constraints (state formulae). So, for example a process-store configuration of the form  $\langle (\mathbf{abs} \ y; \ominus c) P, \ominus d \rangle$  may appear in the transitions of the symbolic semantics.

**Symbolic Reductions.** The internal and observable *symbolic* transitions  $\longrightarrow_s$ ,  $\Longrightarrow_s$  are defined as in Table 1 with  $\vdash$  replaced by  $\vdash_T$  and with  $R_A$  and  $R_O$  replaced by  $R_{As}$  and  $R_{Os}$  in Table 2 resp.

The rule  $R_{As}$  represents with the temporal constraint  $\forall \vec{x}(e \Rightarrow e')$  the substitutions that its operational counterpart  $R_A$  would induce. Notice that in the reduction of  $P$  the variables  $\vec{x}$  in  $e$  are hidden, via existential quantification, to avoid clashes with those in  $P$ .

The function  $F_s$  in  $R_{Os}$  is similar to its operational counterpart  $F$  in Definition 7. However,  $F_s$  records the final global and local stores as well as abstraction guards as past information. As explained before, this past information is needed in next time unit.

$R_{As} \frac{\langle P, \exists \vec{x} e \rangle \longrightarrow_s \langle Q, e'' \wedge \exists \vec{x} e \rangle}{\langle (\mathbf{abs} \ \vec{x}; e') P, e \rangle \longrightarrow_s \langle (\mathbf{abs} \ \vec{x}; e') Q, e \wedge \forall \vec{x}(e' \Rightarrow e'') \rangle}$
$R_{Os} \frac{\langle P, e \rangle \xrightarrow{*}_s \langle Q, e' \rangle \not\rightarrow_s}{P \xrightarrow{(e, e')}_s F_s(Q, e')}$

**Table 2.** Symbolic Rules for Internal and Observable Transitions. The function  $F_s$  is given in Definition 12.

**Definition 12.** Let  $F_s$  be a partial function from configurations to processes defined by  $F_s(P, e) = \mathbf{tell}(\ominus e) \parallel F'(P)$  where:

$$F'(P) = \begin{cases} \mathbf{skip} & \text{if } P = \mathbf{skip} \\ (\mathbf{abs} \ \vec{x}; \ominus e) F'(Q) & \text{if } P = (\mathbf{abs} \ \vec{x}; e) Q \\ F'(P_1) \parallel F'(P_2) & \text{if } P = P_1 \parallel P_2 \\ (\mathbf{local} \ \vec{x}; \ominus e) F'(Q) & \text{if } P = (\mathbf{local} \ \vec{x}; e) Q \\ Q & \text{if } P = \mathbf{next} Q \\ Q & \text{if } P = \mathbf{unless} \ c \ \mathbf{next} Q \end{cases}$$

Clearly, no infinite sequence of internal transitions  $\gamma_1 \longrightarrow_s \gamma_2 \longrightarrow_s \dots$  can be generated by the symbolic semantics.

We now define the input-output behavior and equivalence for the symbolic reduction semantics.

**Definition 13 (Symbolic Relations).** Let  $e$  and  $e'$  be future-free formulae. We write  $e \succeq e'$  whenever  $e \vdash_T e'$ . If  $e \succeq e'$  and  $e' \succeq e$  we write  $e \approx e'$ . If  $e \succeq e'$  and  $e \not\approx e'$  then we write  $e \succ e'$ . We extend  $\succeq$ ,  $\succ$  and  $\approx$  to sequences of future-free formulae:  $w \succeq v$  ( $w \succ v$ ,  $w \approx v$ ) iff for all  $i > 0$ ,  $w(i) \succeq v(i)$  ( $w(i) \succ v(i)$ ,  $w(i) \approx v(i)$ ).

If  $P = P_1 \xrightarrow{(e_1, e'_1)}_s P_2 \xrightarrow{(e_2, e'_2)}_s \dots$ , we write  $P \xrightarrow{(w, w')}_s$  where  $w = e_1.e_2\dots$  and  $w' = e'_1.e'_2\dots$ . The symbolic input-output behavior  $io_s(P)$  is defined as the set  $\{(w, w') \mid P \xrightarrow{(w, w')}_s\}$  modulo  $\approx$ . We write  $P \sim_s^{io} Q$  iff  $io_s(P) = io_s(Q)$ .

As shown in [23], the symbolic reduction semantics and the SOS coincide for the fragment of abstracted-unless free processes:

**Definition 14 (Abstracted-unless free Processes).** We say that  $P$  is abstracted-unless free if it has no occurrences of processes of the form **unless**  $c$  **next**  $Q$  where  $c$  or  $Q$  has occurrences of variables under the scope of an abstraction.

The correspondence is confined to this fragment of the calculus due to the problem of representing the negation of entailment as logic formulae (see [23] for further details).

**Theorem 4 (Semantic Correspondence [23]).** Let  $P$  be an abstracted-unless free process. Suppose that  $P \xrightarrow{(c_1, d_1)} P_1 \xrightarrow{(c_2, d_2)} \dots \xrightarrow{(c_i, d_i)} P_i$  and  $P \xrightarrow{(c_1, e_1)} P_1' \xrightarrow{(c_2, e_2)} \dots \xrightarrow{(c_i, e_i)} P_i'$ . Then for every  $c \in \mathcal{C}$  and  $j \in \{1, \dots, i\}$ ,  $d_j \vdash c$  iff  $e_j \vdash_T c$ .

## 5.2 Monotonic fragment and closure operators

In this section we show that  $io_s(P)$  is a *partial* closure operator—see Section 2. Notice that the “unless” operator unlike the other constructs in **utcc** exhibits non-monotonic input-output behavior in the following sense: Given  $w \succeq w'$  and  $P = \mathbf{unless} \ c \ \mathbf{next} \ Q$ , if  $(w, v), (w', v') \in io_s(P)$ , it may be the case that  $v' \succeq v$ . E.g., take  $Q = \mathbf{tell}(d)$ ,  $w = c.\mathbf{true}^\omega$  and  $w' = \mathbf{true}^\omega$ .

**Definition 15 (Monotonic Processes).** We say that  $P$  is a monotonic process iff  $P$  does not have occurrences of processes of the form **unless**  $c$  **next**  $Q$ .

In fact, for a monotonic process  $P$ ,  $io_s(P)$  is a closure operator. The next proposition basically follows from the corresponding closure-operator properties for internal symbolic transition which in turn can be proven by induction on the (depth of) derivation of internal symbolic transitions.

**Proposition 3 (Closure Properties).** Let  $P$  be an arbitrary **utcc** process. We have the following:

- **Extensiveness:** If  $(w, w') \in io_s(P)$  then  $w' \succeq w$
- **Idempotence:** If  $(w, w') \in io_s(P)$  then  $(w', w') \in io_s(P)$

- **Monotonicity:** if  $P$  is monotonic,  $(w_1, w_2) \in io_s(P)$  and  $w'_1 \succeq w_1$ , then there exists  $w'_2$  such that  $(w'_1, w'_2) \in io_s(P)$  and  $w'_2 \succeq w_2$ .

A pleasant property of closure operators is that they are uniquely determined by its set of fixed points; here referred to as *strongest postcondition*.

**Definition 16 (Strongest Postcondition).** Given a *utcc* process  $P$ , the set

$$sp_s(P) = \{w \mid (w, w) \in io_s(P)\}$$

denotes the strongest postcondition of  $P$ . Moreover, if  $w \in sp_s(P)$ , we say that  $w$  is a *quiescent sequent* for  $P$ , i.e.  $P$  under input  $w$  cannot add any information whatsoever. Define  $P \sim_s^{sp} Q$  iff  $sp_s(P) = sp_s(Q)$ .

Recall that the symbolic semantics transfers the final store of a time-unit to the next one as a past formula (Definition 12). Therefore, for any process  $P$ , if  $s \in sp_s(P)$  then  $s$  is a past-monotonic sequence in the following sense.

**Definition 17 (Past-Monotonic Sequences, PM).** We say that an infinite sequence of future-free formulae  $w$  is past-monotonic iff for all  $i > 1$ ,  $w(i) \vdash_T \ominus w(i-1)$ . The set of infinite sequences of past-monotonic formulae is denoted by  $PM$ .

Before relating the input-output behavior with the strongest postcondition of a process, we need to introduce the following notation.

**Notation 4** The upper closure of a future-free formula  $e$  is the set  $\{e' \mid e' \succeq e\}$  and we write  $\uparrow e$ . We extend this notion to sequences by decreeing that  $\uparrow w = \{w' \mid w' \succeq w\}$ .

The input-output behavior of a monotonic process can be retrieved from its strongest postcondition. This characterization is expressed by the following corollary, whose proof is standard, given that  $io_s(P)$  is a closure operator.

**Corollary 3.** Let  $min$  be the minimum function wrt the order induced by  $\succeq$ . Given a monotonic *utcc* process  $P$ ,

$$(w, w') \in io_s(P) \text{ iff } w' = min(sp_s(P) \cap \uparrow w)$$

Therefore to characterize the input-output behavior of  $P$ , it suffices to specify  $sp_s(P)$ . In the next section we give a denotational semantics  $\llbracket \cdot \rrbracket$  that aims at specifying  $sp_s(\cdot)$  compositionally.

### 5.3 Denotational Model

This section presents a denotational model for the strongest postcondition of *utcc* processes. This semantics is built on the closure operator semantics for *tcc* in [24]. We will briefly discuss the technical problems in giving a closure operator semantics for the abstraction process.

We need the following notations.

**Notation 5** Given the sequence of variables  $\vec{x} = x_1, \dots, x_n$ ,  $\exists_{\vec{x}}$  stands for the expression  $\exists_{x_1} \exists_{x_2} \dots \exists_{x_n}$ . We shall use  $\exists_{\vec{x}} w$  to denote the sequence obtained by pointwise applying  $\exists_{\vec{x}}$  to each constraint in  $w$ . Similarly,  $w \wedge w'$  denotes the sequence  $v$  such that  $v(i) = w(i) \wedge w'(i)$  for  $i > 0$ . Abusing of the notation, for  $\vec{x} \notin \text{fv}(w)$  we understand that none of the variables in  $\vec{x}$  occurs free in the constraints of  $w$ .

The equations for the local and abstraction operators involve the notion of  $\vec{x}$ -variants. Basically, two formulae (or sequences of formulae) are  $\vec{x}$ -variant if they are the same except for the information about the variables in  $\vec{x}$ . Formally:

**Definition 18 ( $\vec{x}$ -variant).** We say that  $e$  and  $e'$  are  $\vec{x}$ -variants if  $\exists_{\vec{x}} e = \exists_{\vec{x}} e'$ . Similarly, the sequence  $w$  is an  $\vec{x}$ -variant of the sequence  $w'$  iff  $\exists_{\vec{x}} w = \exists_{\vec{x}} w'$ .

The denotational semantics is defined as a function  $\llbracket \cdot \rrbracket$  which associates to each process a set of infinite sequences of past-monotonic formulae, namely  $\llbracket \cdot \rrbracket : \text{Proc} \rightarrow \mathcal{P}(\text{PM})$ . The definition of  $\llbracket \cdot \rrbracket$  is given in Figure 1. Recall that  $\llbracket P \rrbracket$  is meant to capture the quiescent sequences of  $P$ ; those sequences  $P$  cannot add any information whatsoever—i.e., the strongest postcondition of  $P$ .

Let us first give some intuition about the semantics of processes that do not bind variables. So, **skip** cannot add any information to any sequence in  $\text{PM}$  ( $\text{D}_S$ ). The sequences to which **tell**( $c$ ) cannot add information are those whose first element can entail  $c$  ( $\text{D}_T$ ). A sequence is quiescent for  $P \parallel Q$  if it is for  $P$  and  $Q$  ( $\text{D}_P$ ). Process **next**  $P$  has no influence in the first element of a sequence, thus  $e.w$  is quiescent for it if  $w$  is quiescent for  $P$  ( $\text{D}_N$ ). A similar explanation can be given for the process **unless**  $c$  **next**  $P$  ( $\text{D}_U$ ). A sequence is quiescent for  $!P$  if every suffix of it is quiescent for  $P$  ( $\text{D}_R$ ).

We now consider the binding processes. A sequence  $w$  is quiescent for  $Q = (\text{local } \vec{x}; c) P$  if there exists an  $\vec{x}$ -variant  $w'$  of  $w$  s.t.  $w'$  is quiescent for  $P$ . Hence, if  $P$  cannot add any information to  $w'$  then  $Q$  cannot add any information to  $w$ . To see this notice that  $w$  and  $w'$  are  $\vec{x}$ -variants; i.e., they are the same except possibly on the information about the variables in  $\vec{x}$ . Clearly  $Q$  cannot add any information on (the global variables)  $\vec{x}$  appearing in  $w$ . So, if  $Q$  were to add information to  $w$ , then  $P$  could also do the same to  $w'$ . But the latter is not possible since  $w'$  is quiescent for  $P$ .

Now, we may then expect that the semantics for the abstraction can be straightforwardly obtained in a similar fashion by quantifying over all possible  $\vec{x}$ -variants. Nevertheless, this is not the case as we shall illustrate below.

**Denotation of Abstraction Using  $\vec{x}$ -variants** Recall that the *ask tcc* process **when**  $c$  **do**  $Q$  is a shorthand for the empty abstraction process **(abs**  $\epsilon$ ;  $c$ )  $Q$  (Notation 1). Recall also that  $\mathcal{T}$  denotes the set of all terms in the underlying constraint system. The first intuition for the denotation of the process  $P = (\text{abs } \vec{x}; c) Q$  is given with the following equation.

$$\bigcap_{\vec{t} \in \mathcal{T}^{|\vec{x}|}} \llbracket (\text{when } c \text{ do } Q) [\vec{t}/\vec{x}] \rrbracket$$

where  $\llbracket \mathbf{when} \ c \ \mathbf{do} \ Q \rrbracket$  is the usual denotational equation for the *ask* processes in **tcc**, i.e.

$$\llbracket \mathbf{when} \ c \ \mathbf{do} \ Q \rrbracket = \{w \mid w(1) \vdash_T c \text{ implies } w \in \llbracket Q \rrbracket\}$$

This equation arises directly from the fact that  $P$  can be viewed as the (possibly infinite) parallel composition of the processes  $(\mathbf{when} \ c \ \mathbf{do} \ Q)[\vec{t}/\vec{x}]$  for every sequence of terms  $\vec{t} \in \mathcal{T}^{|\vec{x}|}$ .

Nevertheless we can give a denotational equation for abstraction which is analogous to that of the local operator. By using the notion of  $\vec{x}$ -variants the equation does not appeal to substitution as the one above. As illustrated below the denotation of abstraction is not entirely dual to the denotation of the local operator. The lack of duality between  $D_L$  and  $D_A$  is reminiscent of the result in CCP [11] stating that negation does not correspond exactly to the complementation (See [9, 11]).

**Notation 6** Given a sequence  $e_1.e_2\dots$ , we use  $\overline{e_1.e_2\dots}$  to denote the past-monotonic sequence

$$e_1.(e_2 \wedge \ominus e_1).(e_3 \wedge \ominus e_2 \wedge \ominus^2 e_1)..$$

*Example 2.* Let  $Q = (\mathbf{abs} \ x; c) P$  where  $c = \mathbf{out}(x)$  and  $P = \mathbf{tell}(\mathbf{out}'(x))$ . Take the past-monotonic sequence

$$w = \overline{(\mathbf{out}(0) \wedge \mathbf{out}'(0)). \mathbf{true}^\omega} \in sp_s(Q).$$

Suppose that we were to give the following definition of  $\llbracket Q \rrbracket$ :

$$\{w \mid \text{for every } x\text{-variant } w' \text{ of } w \text{ if } w'(1) \vdash_T c \text{ then } w' \in \llbracket P \rrbracket\}$$

Let 0 be a term. Notice that we have an  $x$ -variant

$$w' = \overline{(\mathbf{out}(0) \wedge \mathbf{out}'(0) \wedge \mathbf{out}(x)). \mathbf{true}^\omega}$$

of  $w$  s.t.  $w'(1) \vdash_T c$  but  $w' \notin \llbracket P \rrbracket$ . Then  $w \notin \llbracket Q \rrbracket$  under this naive definition of  $\llbracket Q \rrbracket$ .

We fix the above definition in Equation  $D_A$  by using the following condition

$$w' \succeq (\vec{x} = \vec{t})^\omega.$$

for a sequence of terms  $\vec{t}$  s.t.  $|\vec{t}| = |\vec{x}|$  and  $\vec{t} \neq \vec{x}$ . Intuitively, this condition together with  $w'(1) \vdash_T c$  requires that  $w'(1) \vdash_T c \wedge \vec{x} = \vec{t}$  and hence that  $w'(1) \vdash_T c\sigma$  for a substitution  $\sigma = [\vec{t}/\vec{x}]$ . Furthermore  $w' \succeq (\vec{x} = \vec{t})^\omega$  together with  $w' \in \llbracket P \rrbracket$  realizes the operational intuition that  $P$  runs under the substitution  $\sigma$ . In fact the denotation satisfies the following:

**Proposition 4.** Let  $\llbracket \cdot \rrbracket$  as in Figure 1 and  $Q = (\mathbf{abs} \ \vec{x}; c) P$ .

$$w \in \llbracket Q \rrbracket \text{ iff } w \in \bigcap_{\vec{t} \in \mathcal{T}^{|\vec{x}|}} \llbracket (\mathbf{when} \ c \ \mathbf{do} \ P)[\vec{t}/\vec{x}] \rrbracket$$

$D_S[\mathbf{skip}]$	$= PM$
$D_T[\mathbf{tell}(c)]$	$= \{e.w \in PM \mid e \vdash_T c\}$
$D_P[P \parallel Q]$	$= \llbracket P \rrbracket \cap \llbracket Q \rrbracket$
$D_N[\mathbf{next} P]$	$= \{e.w \in PM \mid w \in \llbracket P \rrbracket\}$
$D_U[\mathbf{unless} c \mathbf{next} P]$	$= \{e.w \in PM \mid e \not\vdash_T c \text{ and } w \in \llbracket P \rrbracket\} \cup$ $\{e.w \in PM \mid e \vdash_T c\}$
$D_R[!P]$	$= \{w \in PM \mid \text{for all } v, v' \text{ s.t. } w = v.v',$ $v' \in \llbracket P \rrbracket\}$
$D_L[(\mathbf{local} \vec{x}; c) P]$	$= \{w \in PM \mid \text{there exists an } \vec{x}\text{-variant}$ $w' \text{ of } w \text{ s.t. } w'(1) \vdash_T c \text{ and } w' \in \llbracket P \rrbracket\}$
$D_A[(\mathbf{abs} \vec{x}; c) P]$	$= \{w \in PM \mid \text{for every } \vec{x}\text{-variant } w' \text{ of } w$ $\text{if } w'(1) \vdash_T c \text{ and } w' \succeq (\vec{x} = \vec{t})^\omega$ $\text{for some } \vec{t} \text{ s.t. }  \vec{x}  =  \vec{t}  \text{ and } \vec{x} \neq \vec{t} \text{ then}$ $w' \in \llbracket P \rrbracket\}$

**Fig. 1.** Denotational Semantics for **utcc**. The function  $\llbracket \cdot \rrbracket$  is of type  $Proc \rightarrow \mathcal{P}(PM)$ . In  $D_A$ ,  $\vec{x} = \vec{t}$  denotes the constraint  $\bigwedge_{1 \leq i \leq |\vec{x}|} x_i = t_i$  and  $\vec{t} \neq \vec{x}$  stands for point-wise syntactic difference, i.e.  $\bigwedge_{1 \leq i \leq |\vec{x}|} t_i \neq x_i$  (see Sect. 2). If  $|\vec{x}| = 0$  then  $\vec{x} = \vec{t}$  and  $\vec{x} \neq \vec{t}$  are defined as **true**.

## 5.4 Full-abstraction

In this section we show the correspondence between the symbolic reduction semantics of `utcc` and the denotational one.

*Soundness.* The soundness of the denotation holds for the abstracted-unless free fragment of `utcc` (see Definition 14). The technical reason is that in proving  $sp_s(P) \subseteq \llbracket P \rrbracket$  for  $P = (\mathbf{abs} \ x; c) Q$  where  $x \in fv(Q)$  we need  $Q$  to be monotonic (Definition 15).

**Theorem 5 (Soundness).** *Given an abstracted-unless free process  $P$ ,  $sp_s(P) \subseteq \llbracket P \rrbracket$*

The proof of the above theorem proceeds by induction on the size of  $P$ . Here we only outline the main steps of the abstraction case  $P = (\mathbf{abs} \ \vec{x}; c) Q$ —the other cases proceed as in [22]. As a mean of contradiction assume that  $w \in sp_s(P)$  and  $w \notin \llbracket P \rrbracket$ . By using alpha-conversion we can assume that  $\vec{x} \notin fv(w)$ . Since  $w \notin \llbracket P \rrbracket$  then let  $w'$  be an  $\vec{x}$ -variant of  $w$  s.t.  $w'(1) \vdash_T c$ ,  $w' \succeq (\vec{x} = \vec{t})^\omega$  with  $\vec{x} \neq \vec{t}$  and  $w' \notin \llbracket Q \rrbracket$ . By hypothesis  $w' \notin sp_s(Q)$ . Since  $P$  is abstracted-unless free then  $Q$  is a monotonic process and thus  $Q \xrightarrow{(w', v')}_s$  for a  $v' \succ w'$ . We can now use the facts that  $w' \succeq (\vec{x} = \vec{t})^\omega$  and that  $w'$  is an  $\vec{x}$ -variant of  $w$  to verify that  $(Q \parallel \mathbf{!tell}(\vec{x} = \vec{t})) \xrightarrow{(w', v')}_s$  and thus that  $R = (\mathbf{local} \ \vec{x}) (Q \parallel \mathbf{!tell}(\vec{x} = \vec{t})) \xrightarrow{(w, \exists_{\vec{x}} v')}_s$ . We then show that  $R \sim_s^{io} Q[\vec{t}/\vec{x}]$  and hence  $(w, \exists_{\vec{x}} v') \in io_s(Q[\vec{t}/\vec{x}])$ . Since  $w$  and  $w'$  are  $\vec{x}$ -variants,  $\vec{x} \notin fv(w)$  and  $v' \succ w'$  then  $\exists_{\vec{x}} v' \succ w$ . Therefore,

$$w \notin sp_s(Q[\vec{t}/\vec{x}])$$

Notice that because  $w'(1) \vdash_T c$  and  $w' \succeq (\vec{x} = \vec{t})^\omega$  we have  $w'(1) \vdash_T c[\vec{t}/\vec{x}]$ , and thus  $w(1) \vdash_T c[\vec{t}/\vec{x}]$  since  $w$  and  $w'$  are  $\vec{x}$ -variants. This leads to a contradiction by verifying that for all  $w, \vec{t}$  if  $w \in sp_s(P)$  and  $w(1) \vdash_T c[\vec{t}/\vec{x}]$  it must be the case that  $w \in sp_s(Q[\vec{t}/\vec{x}])$ .

*Completeness.* For completeness we have similar technical problems that in the case of `tcc` [22] namely: the combination between the local and the “unless” operator—see [22] for details. Thus as in [22] the full abstraction is achieved only for the locally independent fragment.

**Definition 19 (Locally Independent Processes).** *We say that  $P$  is locally independent iff  $P$  has no occurrences of processes of the form `unless  $c$  next  $Q$`  under the scope of a local operator.*

The use of the abstracted-unless free condition in the completeness result is analogous to that for soundness.

**Theorem 6 (Completeness).** *Given a locally independent and abstracted-unless free process  $P$ ,  $\llbracket P \rrbracket \subseteq sp_s(P)$*

Similar to the case of soundness, the proof of completeness proceeds by induction on the size of  $P$ . Here we only outline the main steps of the abstraction case  $P = (\mathbf{abs} \ \vec{x}; c) Q$ . As a mean of contradiction assume that  $w \in \llbracket P \rrbracket$  and  $w \notin sp_s(P)$ . By alpha-conversion we can assume that  $\vec{x} \notin fv(w)$ . Since  $w \notin sp_s(P)$  then one can verify that we have a  $\vec{t} \neq \vec{x}$  s.t.  $w(1) \vdash c[\vec{t}/\vec{x}]$  and

$$w \notin sp_s(Q[\vec{t}/\vec{x}]).$$

Let  $w' = w \wedge (\vec{x} = \vec{t})^\omega$ . Notice that  $w'$  is an  $\vec{x}$ -variant of  $w$  because  $\vec{x} \notin fv(w)$ . Thus, since  $w(1) \vdash_T c[\vec{t}/\vec{x}]$  then  $w'(1) \vdash_T c[\vec{t}/\vec{x}]$ . From  $w \in \llbracket P \rrbracket$ , and Equation  $D_A$ ,  $w' \in \llbracket Q \rrbracket$  and by hypothesis  $w' \in sp_s(Q)$ . Using Rule  $R_{As}$  and the fact that  $w \notin sp_s(P)$ , we can show that  $w \notin sp_s(Q)$ . Since  $w' \in sp_s(Q)$  then  $Q \xrightarrow{(w', w')}_s$ . From this we can verify that  $(Q \parallel \mathbf{tell}(\vec{x} = \vec{t})) \xrightarrow{(w', w')}_s$  and  $Q[\vec{t}/\vec{x}] \xrightarrow{(w, w)}_s$ . Therefore,  $w \in sp_s(Q[\vec{t}/\vec{x}])$ , a contradiction.

*Full Abstraction.* From Corollary 3, soundness and completeness, we can derive the following full-abstraction result.

**Corollary 4 (Full abstraction).** *Let  $P$  and  $Q$  be locally independent and monotonic processes and  $\llbracket \cdot \rrbracket$  as in Figure 1. Then*

$$P \sim_s^{io} Q \text{ iff } \llbracket P \rrbracket = \llbracket Q \rrbracket.$$

Therefore, the input-output behavior of the monotonic locally-independent fragment of **utcc** can be compositionally specified in terms of closure operators. To illustrate the applicability of this fragment, in the next section we shall identify a language for security protocols whose semantics can be given in terms of this kind of closure operators.

*Remark 3.* Notice that the connection between the operational semantics and the denotational semantics for well-terminated processes follows from Theorem 4 and the soundness and completeness results in this section.

## 5.5 Application: A security process language

Cryptographic protocols aims at communicating agents securely over an untrusted network. Several process languages have been defined to analyze these protocols [10, 3, 4, 6]. Typically, these calculi provide a mechanism for communication of private names (*nonces*) and they are parametric in an entailment relation over a logic for reasoning about cryptographic properties.

In this section we show how the monotonic locally-independent fragment of **utcc** and its closure operator characterization can be used to give meaning to a process language enjoying the typical features of calculi for security mentioned above. This language arises as specialization of **utcc** with a particular cryptographic constraint systems. We will show that processes in the language can be compositionally specified as closure operators. So, e.g., the set of messages a protocol may produce can be represented as a closure operator over sequences of constraints. The least fixed point of this operator may be used to verify if a secrecy property is not verified by the protocol.

**The modeling language: SCCP** We shall use a syntax of processes following that of SPL [10]. Basically, this language offers primitives to output and receive messages as well as to generate secrets or nonces (randomly-generated unguessable items).

**Definition 20 (Syntax of SCCP).**

$$\begin{array}{ll}
\text{Values } v, v' & ::= n \mid x \\
\text{Keys } k & ::= \text{pub}(v) \mid \text{priv}(v) \\
\text{Messages } M, N & ::= v \mid k \mid X \mid (M, N) \mid \{M\}_k \\
\text{Patterns } \Pi, \Pi' & ::= v \mid k \mid X \mid (\Pi, \Pi') \\
\text{Processes } R & ::= \mathbf{nil} \\
& \quad \mid \mathbf{new}(x)R \\
& \quad \mid \mathbf{out}(M).R \\
& \quad \mid \mathbf{in} [M > \Pi].R \\
& \quad \mid !R \\
& \quad \mid R \parallel R
\end{array}$$

We shall refer to the language given by the above syntax as SCCP (Security Concurrent Constraint Programming Language). The language includes a set of names (values) with  $n, m, A, B$  ranging over it. These values represent ids of principals or nonces. The set of keys is built upon two constructors providing the public ( $\text{pub}(v)$ ) and the private key ( $\text{priv}(v)$ ) associated to a value. Messages can be constructed from composition  $(M, N)$  and encryption  $\{M\}_k$  of messages. As explained below, message decomposition can be obtained by using pattern matching.

Intuitively, processes in SCCP run in time intervals. The output process  $\mathbf{out}(M).R$  broadcasts  $M$  over the network and then it behaves as  $R$  in the next time unit. The input  $\mathbf{in} [M > \Pi].R$  waits for a message  $M$  that matches the pattern  $\Pi$  and binds the variables occurring in the pattern and then it behaves like  $R$  in the next time unit. For example, for every message of the form  $(N, N')_{\text{pub}(A)}$ , the process  $\mathbf{in} [\{M\}_{\text{pub}(A)} > (x, y)].R$  executes  $R[N/x, N'/y]$  in the next time unit. The process  $\mathbf{new}(x)R$  generates a (nonce)  $x$  private to  $R$ . The process  $\mathbf{nil}$  does nothing;  $R \parallel R'$  denotes the parallel execution of  $R$  and  $R'$ . Finally  $!R$  denotes the execution of  $R$  in each time unit.

*Dolev-Yao Constraint System.* Typically in the modeling of security protocols one must take into account all possible actions the attacker may perform. This attacker is usually given in terms of the Dolev and Yao thread model [14] which presupposes an attacker that can eavesdrop, disassemble, compose, encrypt and decrypt messages with available keys. It also presupposes that cryptography is unbreakable.

Before giving a closure operator semantics to our security language, we then need a constraint system handling the cryptographic constructs (e.g. encryption, message composition, etc) and whose entailment relation follows the inferences a Dolev-Yao attacker may perform.

**Definition 21.** Let  $\Sigma_s$  be a signature with constant symbols in  $\mathcal{V}$ , function symbols *enc*, *pair*, *priv* and *pub* and unary predicate *out*. Let  $\Delta_s$  be the closure under deduction of  $\{ F \mid \vdash_s F \}$  with  $\vdash_s$  as in Table 3. The (secure) constraint system is the pair  $(\Sigma_s, \Delta_s)$ .

PRJ	$\frac{F \vdash_s \text{out}((m_1, m_2)) \quad i \in \{1, 2\}}{F \vdash_s \text{out}(m_i)}$
PAIR	$\frac{F \vdash_s \text{out}(m_1) \quad F \vdash_s \text{out}(m_2)}{F \vdash_s \text{out}((m_1, m_2))}$
ENC	$\frac{F \vdash_s \text{out}(m) \quad F \vdash_s \text{out}(k)}{F \vdash_s \text{out}(\{m\}_k)}$
DEC	$\frac{F \vdash_s \text{out}(\text{priv}(k)) \quad F \vdash_s \text{out}(\{m\}_{\text{pub}(k)})}{F \vdash_s \text{out}(m)}$

**Table 3.** Security constraint system entailment relation.

Intuitively,  $\mathcal{V}$  represents the set of principal ids, nonces and values. We use  $\{m\}_k$  and  $(m_1, m_2)$  resp., for *enc*( $m, k$ ) (encryption) and *pair*( $m_1, m_2$ )(composition). Rule ENC says that if one can infer that the message  $m$  as well as a key  $k$  are output on the global channel *out*, then one may as well infer that  $\{m\}_k$  is also output on *out*. The other rules can be explained similarly.

**A Protocol in SCCP** To illustrate the language consider the following simplification of the Denning-Sacco key distribution protocol [13]:

$$\begin{array}{ll} \text{msg}_1 & A \rightarrow B : \{(A, m)\}_{\text{pub}(B)} \\ \text{msg}_2 & B \rightarrow A : \{n\}_{\text{pub}(m)} \end{array}$$

This protocol involves two principals  $A$  and  $B$ . In the first step,  $A$  sends to  $B$  the message  $\{(A, m)\}_{\text{pub}(B)}$  representing the composition of the  $A$ 's identifier and the nonce  $m$  encrypted with the  $B$ 's public key. With its private key,  $B$  decrypts the message sent by  $A$  and he sends a new nonce  $n$  encrypted with the public key generated from  $m$ . The property that must be verified is that only  $A$  and  $B$  can know  $n$ .

Assume the following execution of the protocol between  $A, B$  and  $C$ . Here  $C$  is an intruder, i.e. a malicious agent playing the role of a principal in the protocol.

$$\begin{array}{ll} \text{msg}_1 & A \rightarrow C : \{(A, m)\}_{\text{pub}(C)} \\ \text{msg}'_1 & C \rightarrow B : \{(A, m)\}_{\text{pub}(B)} \\ \text{msg}_2 & B \rightarrow A : \{n\}_{\text{pub}(m)} \end{array}$$

In this execution, the attacker replies to  $B$  the message sent by  $A$  and  $B$  believes that he is establishing a session key with  $A$ . Since the attacker knows the nonce  $m$  from the first message, he can decrypt the message  $\{n\}_{pub(m)}$  and  $n$  is not longer a secret between  $A$  and  $B$  as intended.

We model the behavior of the initiator and the responder in our running example as follows:

$$\begin{aligned}
Init(A, B) &\equiv ! \mathbf{new}(m) \\
&\quad \mathbf{out}(\{(A, m)\}_{pub(B)}) . \mathbf{nil} \\
Resp(B) &\equiv ! \mathbf{in} [M > \{(x, u)\}_{pub(B)}]. \\
&\quad \mathbf{new}(n) \\
&\quad \mathbf{out}(\{n\}_{pub(u)}) . \mathbf{nil} \\
Spy &\equiv \parallel_{A \in \mathcal{P}} ! \mathbf{out}(A) . \mathbf{nil} \\
&\quad \parallel_{A \in \mathcal{P}} ! \mathbf{out}(pub(A)) . \mathbf{nil} \\
&\quad \parallel_{A \in Bad} ! \mathbf{out}(priv(A)) . \mathbf{nil}
\end{aligned}$$

The process  $Spy$  corresponds to the initial knowledge the attacker has. Given the set of principals of the protocol  $\mathcal{P}$ , the spy knows all the names of the principals in the protocol and their public keys. He also knows a set of private keys denoted by  $Bad$ . This set represents the leaked keys, for example, the private key of  $C$  in the above configuration exhibiting the secrecy flaw.

Notice that the processes  $Init$  and  $Resp$  are replicated. This models the fact that principal may initiate different sessions during the execution of the protocol.

**Closure Operators for SCCP** The following definition interprets SCCP processes as monotonic  $utcc$  processes.

**Definition 22.** Let  $I$  be a function from SCCP to monotonic  $utcc$  processes defined by:

$$I = \begin{cases} \mathbf{skip} & \text{if } R = \mathbf{nil} \\ (\mathbf{local } x) I(R') & \text{if } R = \mathbf{new}(x)R' \\ ! \mathbf{tell}(\mathbf{out}(M)) \parallel \mathbf{next } I(R') & \text{if } R = \mathbf{out}(M).R' \\ (\mathbf{abs } \vec{x}; c) \mathbf{next } I(R') & \text{if } R = \mathbf{in} [M > \Pi].R' \\ \parallel_{i \in I} I(R_i) & \text{if } R = \parallel_{i \in I} R_i \\ ! I(R') & \text{if } R = ! R' \end{cases}$$

where  $\vec{x} = fv(\Pi)$  and  $c = \mathbf{out}(M) \wedge (M = \Pi)$ .

It is easy to see that the above interpretation realizes the behavioral intuition of SCCP given before. Intuitively the output  $\mathbf{out}(M)$  is mapped to a process adding the constraint  $\mathbf{out}(M)$ . Since the final store in  $utcc$  is not automatically transferred to the next time interval, the process  $\mathbf{tell}(\mathbf{out}(M))$  is replicated. This reflects the fact that the attacker can remember all the messages posted over the network.

For the case of inputs, we use the guard of the abstraction operator to check if the input received matches with the pattern  $\Pi$ . Take for example  $Resp(B)$  in

our example. This process is interpreted via  $I$  above as:

$$!(\mathbf{abs} \ x, u; \mathbf{out}(\{M\}_{pub(B)}) \wedge M = (x, w)) \mathbf{next} \ Q$$

where  $Q = I(\mathbf{new}(n)\mathbf{out}(\{n\}_{pub(u)})\mathbf{nil})$ . When a constraint of the form  $\mathbf{out}(\{N, N'\}_{pub(B)})$  can be deduced from the current store, the process  $Q[N/x, N'/y]$  is executed in the next time unit as expected.

The following function maps our security processes into its set of fixed points as specified in Figure 1—i.e., its strongest postcondition.

**Definition 23.** For any SCCP process  $R$  we define  $\llbracket R \rrbracket_{SCCP}$  as  $\llbracket I(R) \rrbracket$  with  $I(\cdot)$  as in Definition 22 and  $\llbracket \cdot \rrbracket$  as in Fig. 1.

Since the interpretation function  $I$  is given in terms of the monotonic fragment of  $\mathbf{utcc}$ , it follows from Section 5.4 that  $\llbracket R \rrbracket_{SCCP}$  corresponds to a closure operator.

*Modeling Secrecy Properties.* We can represent protocols in such a way that potential attacks can be specified as the least fixed point of the closure operators representing them. To detect when the secret created by  $Resp$  is revealed by the attacker, we modify the definition of this process as follows:

$$Resp'(B) \equiv ! \mathbf{in} \ [M > \{(x, u)\}_{pub(B)}]. \\ \mathbf{new}(n)(\mathbf{out}(\{n\}_{pub(u)})\mathbf{nil}) \parallel \\ ! \mathbf{in} \ [Y > n].\mathbf{out}(attack).\mathbf{nil}$$

Intuitively,  $Resp'$  outputs the message  $attack$  when the message  $n$  appears unencrypted on the network, i.e., when  $\mathbf{out}(n)$  can be deduced from the current store.

Recall that  $\mathbf{false}$  is an absorbing element for conjunction and it is the greatest future-free formulae wrt  $\succeq$ . Our approach is then to add the constraint  $\mathbf{false}$  when the message  $attack$  can be read from the network. In terms of the closure operator semantics it implies that if a process  $R$  outputs the message  $attack$ , then the least fixed point of the closure operator representing  $R$  is a sequence whose suffix is the sequence  $\mathbf{false}^\omega$ . More precisely,

**Proposition 5.** Let  $R$  be a SCCP process. Let  $f$  be defined as

$$\llbracket R \rrbracket_{SCCP} \cap \llbracket ! \mathbf{when} \ \mathbf{out}(attack) \ \mathbf{do} \ ! \mathbf{tell}(\mathbf{false}) \rrbracket$$

Therefore,  $I(R) \Downarrow_{attack}$  iff the least-fixed point of the closure operator corresponding to  $f$  takes the form  $w.\mathbf{false}^\omega$

The previous proposition allows us to exhibit the secrecy flaw in our running example. I.e., let  $\mathcal{P} = \{A, B, C\}$  be the set of principal names and  $Bad = \{C\}$  be the set of leaked keys in our previous protocol example. Given the process  $DS = \mathbf{Init}(A, C) \parallel_{X \in \mathcal{P}} Resp'(X)$  and

$$f = \llbracket DS \rrbracket_{SCCP} \cap \llbracket ! \mathbf{when} \ \mathbf{out}(attack) \ \mathbf{do} \ ! \mathbf{tell}(\mathbf{false}) \rrbracket$$

The least fixed point  $v$  of  $f$  takes the form  $v = w.\mathbf{false}^\omega$ .

*Closure Properties of a Spy.* We conclude this section by pointing out that the interpretation of the behavior of protocols as closure operators is a natural one. To see our intuition, let us suppose that  $f$  is a closure operator denoting a SCCP Spy eavesdropping and producing information in the network.

- $f(w) \succeq w$ : The Spy produces new information from the one he obtains.
- If  $w \succeq v$  then  $f(w) \succeq f(v)$ : The more information the Spy gets, the more he will infer.
- $f(f(w)) = f(w)$ : The Spy infers as much as possible from the info he gets.

## 6 Concluding Remarks and Related Work

We showed that unlike the processes of its predecessor `tcc`, `utcc` processes can represent Turing-powerful formalisms. As an application we showed the undecidability of monadic FLTL without function symbols nor equality. We also showed that like for `tcc` processes, `utcc` can be represented as partial closure operators. As an application we identify a language for security in which protocols can be represented as closure operators. The language arose as a specialization of `utcc` in a particular cryptographic constraint system.

*Expressiveness of TCC and FLTL.* The expressivity of CCP-based languages has been explored in [24, 21, 28]. These works show that `tcc` processes are finite-state. The results in [21] also imply that the processes of the extension of `tcc` with arbitrary recursive definitions are not finite-state. Nevertheless these results do not imply that they can encode Turing-expressive formalisms.

There are several works addressing the decidability of fragments of FLTL. In Section 4 we already discussed that in [18] showing that the validity problem in monadic TLV without equality and function symbols is decidable. As mentioned before, TLV unlike TLV-`flex` does not allow quantification over flexible variables. Our decidability results justifies the imposition of this quantification restriction.

The work in [21] shows the decidability of the satisfiability problem restricted to negation-free TLV-`flex` formulae. It was also suggested in [21] that one could dispense with this restriction. This paper refutes this since in the absence of this restriction one can obviously define universal quantification (not present in the negation-free fragment of [21]) and then be able to reproduce the encoding of looping Minsky machines here presented.

Based on the undecidability result of TLV( $\emptyset$ ) [27] (i.e. TLV with the empty set of predicates), [18] proves an incompleteness result for monadic without equality and function symbols TLP logic. Unlike TLV, in TLP the interpretation of the predicates is flexible (state dependent) and all the variables are rigid. [18] also relates undecidability results of  $n$ -adic fragments of TLP with undecidability results of  $n + 1$ -adic fragments of TLV. Thus adding binary predicates turns TLV strongly incomplete. In [16] the *monodic* fragment of FLTL is introduced. A formula is monodic if every subformulae beginning with a temporal operator have at most one free variable. In this case the authors use a TLP-like semantics and

conclude that the set of valid formulae in the 2-variable monadic fragment (i.e. monadic formulae with at most 2 distinct individual variables) is not recursively enumerable even considering finite domains in the interpretation. Nevertheless validity in the fragment of 2-variable monadic formulae is decidable. In [12] these results are extended claiming the undecidability for validity in the monadic monadic 2-variable with equality fragment of TLP.

*Closure-Operator Semantics and Security.* As previously mentioned the closure-operator semantics for `utcc` builds on that for its predecessor `tcc` in [24] and [22]. The denotation in these works map processes into sequences of constraints. Because of the technical difficulties posed by the abstraction operator our denotation maps instead processes into sequences of past-monotonic sequences of future-free FLTL formulae.

Several process languages have been defined to analyze security protocols. For instance Crazzolaro and Winskel's SPL [10], the spi calculus variants by Abadi [3] and Amadio [4], and Boreale's calculus in [6] are all operationally defined in terms of configurations containing items of information (messages) which can only increase during evolution. Such monotonic evolution of information is akin to the notion of monotonic store in CCP. Moreover, the calculi in [4, 6, 15] are parametric in an entailment relation over a logic for reasoning about protocol properties very much like CCP is parametric in an entailment relation over an underlying constraint system.

Although `utcc` can be used to reason about certain aspects of security protocols (e.g., secrecy), it was not specifically designed for this application domain. Here we illustrated how the closure operator semantics of `utcc` may offer new reasoning techniques for the verification of security protocols. We also argued for the closure operators as a natural characterization of the information that can be inferred (e.g., by Spy) from a protocol. Furthermore, the closure operator semantics presented here and the full abstraction theorem in Section 5.4 may allow us to tailor techniques based on behavioral equivalences, e.g. [3, 15], for analyzing security protocols. An example modeling the Needham-Schroeder protocol in `utcc` was presented in [23]. This example however uses non-monotonic processes which do not allow a closure-operator representation. To our knowledge closure operators had not been considered in the study of security protocols.

The successful *logic programming* approach to security protocols in [2] is closely related to ours. Basically, in [2] protocols are modeled as a set of Horn clauses rather than processes. The verification of the secrecy property relies in deducing (or proving that it is not possible) the predicate  $attack(M)$  from the set of Horn clauses. A benefit from our approach is that we can overcome the problem of false attacks pointed in [5]. E.g. Consider a piece data that needs to be kept secret in a first phase of the protocol and later is revealed when is not longer a secret. Because the lack of temporal dependency this may generate a false attack. The temporal approach presented here allows us to control when a message is required to be secret. The work in [5] also avoid false attacks by using a linear logic approach rather than a temporal one.

## References

1. M. Abadi. Corrigendum: The power of temporal proofs. *Theor. Comput. Sci.*, 70(2), 1990.
2. M. Abadi and B. Blanchet. Analyzing Security Protocols with Secrecy Types and Logic Programs. *Journal of the ACM*, 52(1), 2005.
3. M. Abadi and A. D. Gordon. A calculus for cryptographic protocols: The spi calculus. In *Fourth ACM Conference on Computer and Communications Security*, pages 36–47. ACM Press, 1997.
4. R. M. Amadio, D. Lugiez, and V. Vanackère. On the symbolic reduction of processes with cryptographic functions. *Theor. Comput. Sci.*, 290(1), 2003.
5. B. Blanchet. Security Protocols: From Linear to Classical Logic by Abstract Interpretation. *Information Processing Letters*, 95(5), 2005.
6. M. Boreale and M. G. Buscemi. A method for symbolic analysis of security protocols. *Theor. Comput. Sci.*, 338(1-3), 2005.
7. E. Borger, E. Gradel, and Y. Gurevich. *The Classical Decision Problem*. Springer, 2001.
8. J. R. Buchi. On a decision method in restricted second order arithmetic. In *Proc. of Int. Conf. on Logic, Methodology, and Philosophy of Science*, 1962.
9. A. Cortesi, G. Filé, F. Ranzato, R. Giacobazzi, and C. Palamidessi. Complementation in abstract interpretation. *ACM Trans. Program. Lang. Syst.*, 19(1), 1997.
10. F. Crazzolara and G. Winskel. Petri nets in cryptographic protocols. *ipdps*, 03, 2001.
11. F. S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 19(5), 1997.
12. A. Degtyarev, M. Fisher, and A. Lisitsa. Equality and monodic first-order temporal logic. *Studia Logica*, 72(2), 2002.
13. D. E. Denning and G. M. Sacco. Timestamps in key distribution protocols. *Commun. ACM*, 24(8), 1981.
14. M. Fiore and M. Abadi. Computing symbolic models for verifying cryptographic protocols. In *Proc. of the 14th IEEE Workshop on Computer Security Foundations*. IEEE CS, 2001.
15. C. Fournet and M. Abadi. Hiding names: Private authentication in the applied pi calculus. In *Proc. of ISSS*, 2003.
16. I. M. Hodkinson, F. Wolter, and M. Zakharyashev. Decidable fragment of first-order temporal logics. *Ann. Pure Appl. Logic*, 106(1-3), 2000.
17. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.
18. S. Merz. Decidability and incompleteness results for first-order temporal logics of linear time. *Journal of Applied Non-Classical Logics*, 2(2), 1992.
19. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
20. M. L. Minsky. *Computation: finite and infinite machines*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.
21. M. Nielsen, C. Palamidessi, and F. Valencia. On the expressive power of temporal concurrent constraint programming languages. In *Proc. of PPDP'02*. IEEE Computer Society, 2002.
22. M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(1), 2002.

23. C. Olarte and F. D. Valencia. Universal concurrent constraint programming: Symbolic semantics and applications to security. In *Proc. of SAC 2008*. ACM, 2008.
24. V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of LICS'94*. IEEE CS, 1994.
25. V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
26. V. A. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91*. ACM Press, 1991.
27. A. Szalas and L. Holenderski. Incompleteness of first-order temporal logic with until. *Theor. Comput. Sci.*, 57, 1988.
28. F. D. Valencia. Decidability of infinite-state timed ccp processes and first-order ltl. *Theor. Comput. Sci.*, 330(3), 2005.