Available online at www.sciencedirect.com
SCIENCE DIRECT
Electronic Notes in Theoretical Computer Science
Electronic Notes in Theoretical Computer Science 112 (2005) 19–36
www.elsevier.com/locate/entcs

# Abstract Model Checking of tccp programs

María Alpuente [1,2] María del Mar Gallardo [1,3]
Ernesto Pimentel[1,4] Alicia Villanueva [1,5]

Abstract

The *Timed Concurrent Constraint* programming language (tccp) introduces time aspects into the Concurrent Constraint paradigm. This makes tccp especially appropriate to analyze by model checking timing properties of concurrent systems. However, even if very compact state representations are obtained thanks to the use of constraints in tccp, large state spaces can be still generated which may prevent model checking tools from verifying tccp programs completely. In this paper, we introduce an abstract methodology which is based on over– and under–approximating tccp models and mitigates the state explosion problem which is common to traditional model checking algorithms. We ascertain the conditions for the correctness of the abstract technique and show that, due to the timing aspects of the language, this semantics does not correctly simulate the suspension behavior, which is a key feature of tccp. Then, we present a refined abstract semantics which correctly models suspension.

*Keywords:* Model Checking, Timed Concurrent Constraint Programming, Abstract Interpretation.

## 1 Introduction

In the past few years, some extensions of the concurrent constraint paradigm [4,20] have been defined in order to model reactive systems. All these extensions introduce a notion of time which makes it possible to model the typical

[1] This research is supported by the CICYT project TIC2001-2705-C03-02.
[2] Email: alpuente@dsic.upv.es
[3] Email: gallardo@lcc.uma.es
[4] Email: ernesto@lcc.uma.es
[5] Email: villanue@dsic.upv.es

doi:10.1016/j.entcs.2004.01.024

ingredients of these systems, such as timeouts, preemptions, etc. The automatic verification of systems specified in the timed concurrent constraint language tccp of [4] was first studied in [11]. Then, an exhaustive method for applying the classical model checking technique to tccp was proposed in [12] which uses the temporal logic for reasoning about tccp programs of [5]. The main idea behind these methods is to take advantage of the constraint dimension of tccp in order to obtain a compact representation of the system which is then used as an input for the model checking algorithms. Unfortunately, both [11] and [12] develop exhaustive model checking algorithms, which causes the traditional state explosion problem and makes them not applicable to large size systems. In this work, we develop some suitable approximation techniques which are based on abstract interpretation [8] in order to drastically reduce the state space of model checking tccp thus providing a framework where exhaustive analysis of more complicated systems could be achieved.

Abstract model checking [7,10,17] combines abstract interpretation [8] and model checking [6] to improve the automatic verification of large systems. Applying abstract model checking involves the abstraction of both the model to be analyzed ($M$) and the properties to be checked within the model. Usually, in the classic abstract model checking literature, the abstract model $M^+$ is an over-approximation of the original one $M$, meaning that each possible concrete execution trace is mimicked in the abstract model. This approach allows one to verify properties which regard all the possible behavior paths. Two techniques have been successfully developed to construct $M^+$. The *predicate abstraction* approach consists of substituting some selected model expressions with boolean variables, which leads to important simplifications (e.g., this is used in the tool SLAM [3,2]). In contrast, the *data abstraction* method reduces the type of certain data by transforming its original concrete domain into an approximate and simpler domain. For instance, this second approach has been used for abstracting models in the tools Bandera [16] and $\alpha$SPIN [13].

In this paper, we follow the *data abstraction* method to approximate tccp computations. The common way of formalizing this technique is to introduce *abstract operations* that over-approximate the original ones (see, for instance, [14] where a data-based abstraction for the modeling language Promela is developed). However, inaccurate abstract models would be obtained by over-approximation due to the time dimension of tccp. We overcome this problem by combining over- and under-approximation in the abstraction of tccp operators. This approach is novel and allows us to build abstract models which are satisfactorily precise.

The abstraction of models usually involves adding non–determinism, due to the loss of information caused by the abstraction, and does affect the suspension of processes: the suspension of a process in the original model does not generally imply that the process abstractly suspends. This is why abstraction is not adequate for analyzing system deadlocks. In the case of tccp, changing the suspension behavior might have undesirable consequences because processes are totally synchronized meaning that, at each time instant, all enabled agents (i.e., actions) involving a unit time are simultaneously carried out.

We have successively solved these two problems in the paper. First, we have defined an abstract semantics which takes into account the potential non–determinism added by the abstraction. We have proved that, provided the suspension behavior is correctly simulated, the abstract semantics is correct w.r.t. the original one. We have also defined a source-to-source transformation from the tccp program into its abstract version which is valid for this case. Source-to-source transformations are particularly interesting in the context of abstract model checking because they permit reusing model checkers. Next, we have slightly modified the abstract semantics, solving the suspension problem mentioned above. Due to lack of space, the problem of abstracting properties to adapt them to the new model representation is not dealt with in the paper.

The paper is organized as follows. Section 2 recalls the main features of the tccp language. In Section 3, we introduce our data abstraction methodology for tccp. We also present the abstract semantics for tccp, which does not take into account the potential suspension of processes. Section 4 discusses the correctness of this semantics and introduces a refined abstract semantics which correctly models process suspension. Finally, Section 5 shows the conclusions and discusses some future work.

## 2 The tccp language

In [4], the *Timed Concurrent Constraint* language (tccp in short) was defined as an extension of the Concurrent Constraint programming language ccp [19]. In the cc paradigm, the notion of *store as valuation* is replaced by the notion of *store as constraint*. The computational model is based on a global store where constraints are accumulated, and a set of agents which interact with the store. The model is parametric w.r.t. a cylindric constraint system $\mathcal{C}$ defined below. In tccp, a new conditional agent (now $c$ then $A$ else $A$) is introduced (w.r.t. ccp) which makes it possible to model behaviors where the absence of information can cause the execution of a specific action. Intuitively, the execution of a tccp program evolves by asking and telling information to the store. Let us briefly recall the syntax of the language:

$$A ::= \mathsf{stop} \mid \mathsf{tell}(c) \mid \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i \mid \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, A \mid A||A \mid \exists x\, A \mid \mathsf{p}(x)$$

where $c, c_i$ are *finite constraints* (i.e., atomic propositions) of $\mathcal{C}$. A tccp *process* $P$ is an object of the form $D.A$, where $D$ is a set of procedure declarations of the form $\mathsf{p}(x) :: -A$, and $A$ is an agent.[6]

Intuitively, the stop agent finishes the execution of the program, $\mathsf{tell}(c)$ adds the constraint $c$ to the store, whereas the choice agent ($\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$) consults the store and non-deterministically executes the agent $A_i$ in the following time instant, provided the store satisfies the condition $c_i$; otherwise the agent suspends. The conditional agent ($\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$) can detect *negative information* in the sense that, if the store satisfies $c$, then the agent $A$ is executed; otherwise (even if $\neg c$ does not hold), $B$ is executed. $A1||A2$ executes the two agents $A1$ and $A2$ in parallel. The $\exists x\ A$ agent is used to hide the information regarding $x$, i.e., it makes $x$ local to the agent $A$.

The notion of time is introduced by defining a global clock which synchronizes all agents. In the semantics, the only agents which consume time (*timing agents*) are the tell, choice and procedure call agents.

We show an example of a tccp program in Figure 1. This program models a photocopier by means of four procedure declarations which represent the two main processes (`user(C,A)` and `photocopier(C,A,Max,E,T)`) and the synchronization of such processes (`system(Max,E,C,A,T)` and `initialize(Max)`).

Agent `user(C,A)` can execute non-deterministically three different actions or do nothing. The system is assumed to be synchronous, in the sense that the user cannot execute any action (through stream `C`) before the photocopier satisfies the previous request. This fact is modeled by the stream variable `A`.

The stream variable `T` is used to detect if no request has been received after `Max` time units. When this occurs, the photocopier is automatically turned-off. The system is initialized (`initialize(Max)`) and then synchronized by executing in parallel the photocopier and the user processes.

## 3 Abstracting tccp programs

Recently, some model checking algorithms have been formalized for the concurrent constraint paradigm. The common idea behind them is to exploit the constraint nature of the language to represent the model of the system in a compact way. However, the state explosion problem of classical model

[6] We assume that all programs which we consider in this work are well typed.

```
user(C,A)::=ask(A=[free|_]) → tell(C=[on|_]) +
    ask(A=[free|_]) → tell(C=[off|_]) +
    ask(A=[free|_]) → tell(C=[nc|_]) +
    ask(A=[free|_]) → tell(true).
photocopier(C,C',A,A',Max,E,E',T,T')::=∃ Aux,Aux'(
   now (T=[Aux|_] ∧ Aux>0) then
       now (C=[on|_]) then
           tell(E'=[going|_] ∧ T'=[Max|_] ∧ A'=[free|_])
       else now (C=[off|_]) then
               tell(E'=[stop|_] ∧ T'=[Max|_] ∧ A'=[free|_])
           else now (C=[nc|_]) then
                   tell(E'=[going|_] ∧ T'=[Max|_] ∧ A'=[free|_])
               else tell(Aux'=Aux-1) || tell(T'=[Aux'|_] ∧ A'=[free|_])
   else tell(E'=[stop|_]) || tell(A'=[free|_])).
system(Max,E,C,A,T)::=∃ E',C',A',T'(tell(E=[_|E']) || tell(C=[_|C']) ||
   tell(A=[_|A']) || tell(T=[_|T']) || user(C,A) ||
   ask(true)→ask(true)→photocopier(C,C',A,A',Max,T,T',E,E') ||
   ask(A'=[free|_])→system(Max,E',C',A',T')).
initialize(Max)::=∃ E,C,A,T(tell(A=[free|_]) || tell(T=[Max|_]) ||
                           tell(E=[off|_]) || system(Max,E,C,A,T).
```

Figure 1. tccp program modeling a photocopier

checking techniques also happens in these algorithms. In order to avoid this problem, symbolic representations ([1,18]) and abstract models have been proposed ([10,14]), which we combine in this work.

## 3.1 Abstracting constraint systems

**Definition 3.1** A *simple constraint system* is a structure such as $\langle \mathcal{C}, \vdash \rangle$ where $\mathcal{C}$ is the set of atomic constraints and relation $\vdash \subseteq \wp(\mathcal{C}) \times \mathcal{C}$ satisfies

C1. $u \vdash C$, for all $C \in u$. C2. $u \vdash C$, if $u \vdash C', \forall C' \in v$, and $v \vdash C$

Relation $\vdash$ may be extended to the relation $\vdash \subseteq \wp(\mathcal{C}) \times \wp(\mathcal{C})$ as:

$$u \vdash v \iff \forall C \in v, u \vdash C$$

**Proposition 3.2** *Relation $\vdash$ has the following properties:*

(i) *(Reflexivity) $\forall u \in \wp(\mathcal{C}).u \vdash u$.*

(ii) *(Transitivity) $\forall u, v, w \in \wp(\mathcal{C}).u \vdash v, v \vdash w$ implies that $u \vdash w$.*

During tccp program computation, stores are represented by elements of $\wp(\mathcal{C})$. That is, if $u \subseteq \mathcal{C}$ is the current store, the information accumulated by $u$ is the *conjunction* of all constraints $C \in u$. In addition, $\vdash$ is the entailment relation used to deduce information from stores.

An *abstract interpretation* (an *abstraction*) of the simple constraint system $(\mathcal{C}, \vdash)$ is given by defining an *upper closure operator* (uco) $\rho : \wp(\wp(\mathcal{C})) \rightarrow \wp(\wp(\mathcal{C}))$, that is, a *monotonic* ($sst_1 \subseteq sst_2$ then $\rho(sst_1) \subseteq \rho(sst_2)$), *idempotent* ($\rho(sst) = \rho(\rho(sst))$) and *extensive* ($sst \subseteq \rho(sst)$) operator. The intuition of this definition is that each store $st \subseteq \mathcal{C}$ is abstracted by its closure $\rho(\{st\})$. Closures operators have many interesting properties. For instance, when the domain considered is a complete lattice, as $(\wp(\wp(\mathcal{C})), \subseteq)$, each closure operator is uniquely determined by the set of its fixed points. In the context of abstract interpretation, closure operators are important because abstract domains can be equivalently defined by using them or by Galois insertions as introduced in [9]. Let $\iota : \wp(\wp(\mathcal{C})) \rightarrow E$ be an isomorphism. Then, given an uco $\rho : \wp(\wp(\mathcal{C})) \rightarrow \wp(\wp(\mathcal{C}))$, structure $(\wp(\wp(\mathcal{C})), \iota \circ \rho, \iota^{-1}, E)$ is a Galois insertion, where $\iota \circ \rho$ and $\iota^{-1}$ are the abstraction and concretization functions, respectively.

Note that, using abstract interpretation terminology, $\rho(\{st\})$ is the most precise abstraction of the store $st \in \wp(\mathcal{C})$ and if $\rho(\{st\}) \subseteq sst$, then $sst$ is also an abstraction of $st$.

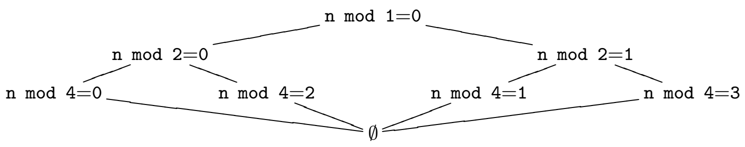

Figure 2. Closure operator $\rho_x$

**Example 3.3** Let $\mathcal{C} = \{x = n, y = m | n, m \in \mathbb{N}\}$, and $\rho_x : \wp(\wp(\mathcal{C})) \rightarrow \wp(\wp(\mathcal{C}))$ an abstraction which does not affect variable $y$, while the abstract value of $x$ is given in Figure 2. In this figure, expression `n mod a = b` represents the set of stores which contain the constraint $x = n$, with $n\ mod\ a = b$. In order to construct $\rho_x$, we define the following sets of stores where $m \in \mathbb{N}$: $[y = m] = \{\{y = m\}\}$, $[n\ mod\ a = b, y = m] = \{\{x = b + ak, y = m\} | k \in \mathbb{N}\}$ and $[n\ mod\ a = b] = \{\{x = b + ak\} | k \in \mathbb{N}\}$

Using the lub operator induced by the lattice shown in Figure 2 (denoted below as $\bigsqcup$), we define operator $\bigsqcup_x$ over these sets as:

- $[n\, mod\, a_1 = b_1, y = m] \bigsqcup_x [n\, mod\, a_2 = b_2, y = m] = [(n\, mod\, a_1 = b_1) \bigsqcup (n\, mod\, a_2 = b_2), y = m]$.
- $[n\, mod\, a_1 = b_1] \bigsqcup_x [n\, mod\, a_2 = b_2] = [(n\, mod\, a_1 = b_1) \bigsqcup (n\, mod\, a_2 = b_2)]$.
- $[c_1] \bigsqcup_x [c_2] = [c_1] \cup [c_2]$, otherwise.

Now, $\rho_x$ is defined as: $\rho_x(\emptyset) = \emptyset$; $\rho_x(\{st\}) = [c]$ iff $[c]$ is the smallest set of stores such that $st \in [c]$; $\rho_x(\{st_i | i \in I\}) = (\bigsqcup_x)_{i \in I} \rho_x(\{st_i\})$.

The following definition introduces two dual entailment relations into the abstract constraint systems.

**Definition 3.4** Given a simple constraint system $\langle \mathcal{C}, \vdash \rangle$ and an abstraction $\rho : \wp(\wp(\mathcal{C})) \to \wp(\wp(\mathcal{C}))$, we define the *over-* and *under-approximated constraint systems* $\langle \wp(\mathcal{C}), \vdash^+_\rho \rangle$ and $\langle \wp(\mathcal{C}), \vdash^-_\rho \rangle$ where $\vdash^+_\rho, \vdash^-_\rho \subseteq \wp(\wp(\mathcal{C})) \times \wp(\wp(\mathcal{C}))$ are:

- $sst_1 \vdash^+_\rho sst_2 \iff \exists u \in \rho(sst_1), \exists v \in sst_2$ such that $u \vdash v$.
- $sst_1 \vdash^-_\rho sst_2 \iff \forall u \in \rho(sst_1), \exists v \in sst_2, u \vdash v$.

The following proposition justifies the names of the new structures given in the previous definition.

**Proposition 3.5** *Given a simple constraint system $(\mathcal{C}, \vdash)$ and an abstraction $\rho : \wp(\wp(\mathcal{C})) \to \wp(\wp(\mathcal{C}))$, then*

*1. If $u \vdash v$, then $\rho(\{u\}) \vdash^+_\rho \{v\}$.* *2. If $\rho(\{u\}) \vdash^-_\rho \{v\}$, then $u \vdash v$*

**Proposition 3.6** *Given a simple constraint system $\langle \mathcal{C}, \vdash \rangle$ and an abstraction $\rho : \wp(\wp(\mathcal{C})) \to \wp(\wp(\mathcal{C}))$, then*

(i) *(Reflexivity for $\vdash^+_\rho$) $\forall sst \in \wp(\wp(\mathcal{C})).sst \vdash^+_\rho sst$.*

(ii) *(Transitivity for $\vdash^-_\rho$) $\forall sst_1, sst_2, sst_3 \in \wp(\wp(\mathcal{C})).sst_1 \vdash^-_\rho sst_2, sst_2 \vdash^-_\rho sst_3$ implies that $sst_1 \vdash^-_\rho sst_3$.*

Intuitively, the image associated to a store by means of $\vdash^+_\rho$ is bigger than the one obtained by applying $\vdash^-_\rho$ to the same store. However, it is worth noting that, in general, relation $\vdash^+_\rho$ is not transitive and $\vdash^-_\rho$ is not reflexive, as shown in the following example. This means that abstract entailment relations lose information with respect to the original relation as usual when abstraction is applied. In addition, each abstract relation loses only one of the two properties characterizing the entailment relations. This fact follows from the definition of the relations.

**Example 3.7** Assume that the constraint system of Example 3.3 is extended with the constraint $n \; mod \; 2 = 0$ containing the natural meaning about value $n$ of variable $x$.

(i) $\vdash^+_\rho$ is not transitive. $\{\{x = 8\}\} \vdash^+_{\rho_x} \{\{n \; mod \; 2 = 0\}\}$ and $\{\{n \; mod \; 2 = 0\}\} \vdash^+_{\rho_x} \{\{x = 6\}\}$, since $\{x = 6\} \in \rho_x(\{\{n \; mod \; 2 = 0\}\})$ and $\{x = 6\} \vdash \{x = 6\}$. However, $\{\{x = 8\}\} \nvdash^+_{\rho_x} \{\{x = 6\}\}$.

(ii) $\vdash^-_\rho$ is not reflexive. $\{\{x = 2\}\} \nvdash^-_{\rho_x} \{\{x = 2\}\}$, since $\{x = 6\} \in \rho_x(\{\{x = 2\}\})$ and $\{x = 6\} \nvdash \{x = 2\}$.

**Definition 3.8** Let us define the operator $\sqcup^\rho : \wp(\wp(\mathcal{C})) \rightarrow \wp(\wp(\mathcal{C}))$ as $sst_1 \sqcup^\rho sst_2 = \rho(\{u \cup v | u \in sst_1, v \in sst_2, u \cup v \vdash true\})$.

Operator $\sqcup^\rho$ provides the abstract information given by the accumulation of the constraints in its two operands. Note that, following the operational semantics of the language, inconsistent stores might appear (due to the tell agent), thus it is necessary to explicitly remove them from $sst_1 \sqcup^\rho sst_2$ as is done by means of the operator $\sqcup^\rho$. The following proposition states that operator $\sqcup^\rho$ correctly approximates $\cup$.

**Proposition 3.9** *For all $u, v \in \wp(\mathcal{C})$ if $u \cup v \vdash true$ then $\rho(\{u \cup v\}) \subseteq \rho(\{u\}) \sqcup^\rho \rho(\{v\})$*

In tccp, special constraint systems called *cylindric constraint systems* are used. Cylindric constraint systems are defined as follows.

**Definition 3.10** $\langle \mathcal{C}, \vdash, Var, \exists \rangle$ is a *cylindric constraint system* iff $\langle \mathcal{C}, \vdash \rangle$ is a simple constraint system, $Var$ is a denumerable set of variables, and for each $x \in Var$, there exists a function $\exists_x : \wp(\mathcal{C}) \rightarrow \wp(\mathcal{C})$ such that for each $u, v \in \wp(C)$:

(i) $u \vdash \exists_x u$

(ii) $u \vdash v$ then $\exists_x u \vdash \exists_x v$

(iii) $\exists_x(u \cup \exists_x v) = \exists_x u \cup \exists_x v$

(iv) $\exists_x(\exists_y u) = \exists_y(\exists_x u)$

A *set of diagonal elements* for a cylindric constraint system is a family $\{\delta_{xy} \in \mathcal{C} | x, y \in var\}$ such that

(i) $\emptyset \vdash \delta_{xx}$

(ii) If $y \neq x, z$ then $\delta_{xz} = \exists_x(\delta_{xy} \cup \delta_{yz})$.

(iii) If $x \neq y$ then $\delta_{xy} \cup \exists_x(v \cup \delta_{xy}) \vdash v$.

Diagonal elements allow us to hide variables, representing local variables, as well as to implement parameter passing among predicates. Thus, quantifier $\exists_x$ and diagonal elements $\delta_{xy}$ allow us to properly deal with variables in constraint systems. Assuming that the original constraint system $\langle \mathcal{C}, \vdash \rangle$ to be abstracted is a cylindric constraint system, and given an abstraction $\rho : \wp(\wp(\mathcal{C})) \rightarrow \wp(\wp(\mathcal{C}))$, in general, the over and under-approximated constraint systems $\langle \wp(\mathcal{C}), \vdash^+_\rho \rangle$ and $\langle \wp(\mathcal{C}), \vdash^-_\rho \rangle$ are not cylindric constraint systems. Example 3.7 shows that some property of the underlying simple constraint system may be lost during the abstraction process. In addition, the remaining properties concerning the existential quantifier or the diagonal elements may also be lost. An extensive study of the conditions that the abstraction $\rho$ has to satisfy for the abstraction process to preserve all these properties can be found in [15] where a generalized semantics for logic concurrent languages is introduced. In short, $\rho$ must satisfy some consistency properties to ensure that the existential quantification behaves correctly. We extend function $\exists_x$ to sets of stores as $\exists_x : \wp(\wp(\mathcal{C})) \to \wp(\wp(\mathcal{C}))$ where $\exists_x sst = \{\exists_x u | u \in sst\}$.

**Example 3.11** Consider the tccp program modeling a photocopier shown in Figure 1. Let $\mathcal{C}$ be the set of atomic constraints appearing in the code. Define the set $msg = \{on, off, nc\}$. Given $X, X' \in Var$, construct the sets $msg(X, X') = \{X = [A|X'] | A \in msg\}$ and $MSG = \cup_{X,X' \in Var} msg(X, X')$. Divide each store $u \in \wp(\mathcal{C})$ into the subsets $u_1 = u - MSG$ and $u_2 = u \cap MSG$. Then the upper closure operator $\rho : \wp(\wp(\mathcal{C})) \to \wp(\wp(\mathcal{C}))$ is defined as follows.

- Given $u \in \wp(\mathcal{C})$, then $u' \in \rho(\{u\})$ if $u' = u_1 \cup u'_2$ where $u'_2$ is constructed by choosing an element $C' \in msg(X, X')$ whenever $msg(X, X') \cap u \neq \emptyset$.
- Given $u \in \wp(\mathcal{C})$, then $\rho(\{u\}) = \{u\}$ iff $\forall X, X' \in Var, msg(X, X') \cap u = \emptyset$.
- $\rho(sst) = (\cup_{u \in sst} \rho(\{u\}))$.

In short, function $\rho$ abstracts the messages in $msg$. For instance,

$$\rho(\{\{X = [on|X']\}\}) = \{\{X = [off|X']\}, \{X = [on|X']\}, \{X = [nc|X']\}\}$$

Note that an implementation of this abstraction would substitute the concrete constants *on*, *off* and *nc* by an abstract constant (for example, *msg*), thus making the abstract store simpler.

## *3.2 Abstract Semantics*

The source-to-source transformation from the original model into the abstract one is a well-known technique of integrating abstraction and model checking [13,16] since it enables reusing the existing model checkers for the original language. In this section, we study the difficulties of applying this method to tccp programs.

In the case of tccp, the abstraction of now has to be done with special care. The reason for this is that the non-determinism introduced when abstracting this agent cannot be handled in tccp instantaneously. The use of ask involves passing one time unit. To solve this problem, we have introduced a new agent $\text{ask}^\alpha$ which allows us to introduce non-determinism without consuming time.

We formalize the abstract operational semantics of a tccp model (with the new agent) in terms of a transition relation similar to the operational semantics of the original tccp language. Similarly to the original operational semantics of tccp, each transition involves time passing. However, we will show that the time aspects of tccp may impede the correct simulation of the synchronization of agents in the concrete model. This aspect differentiates tccp from other modeling languages having no time aspects.

$$\textbf{R1}\quad \langle \mathsf{stop}, sst\rangle \not\longrightarrow \qquad\qquad \textbf{R2}\quad \langle \mathsf{tell}(c), sst\rangle \longrightarrow \langle \emptyset, sst \sqcup c\rangle$$

$$\textbf{R3}\quad \langle \sum_{i=0}^{n} \mathsf{ask}(c_i) \to A_i, sst\rangle \longrightarrow \langle A_j, sst\rangle \quad \text{if } 0 \le j \le n, \text{ and } sst \vdash^{+} c_j$$

$$\textbf{R4}\quad \frac{\langle A_j, sst\rangle \longrightarrow \langle A'_j, sst'\rangle}{\langle \sum_i^n \mathsf{ask}^{\alpha}(c_i) \to A_i, sst\rangle \longrightarrow \langle A'_j, sst'\rangle} \quad \text{if } sst \vdash^{+} c_j$$

$$\textbf{R5}\quad \frac{\langle B1, sst\rangle \longrightarrow \langle B1', sst'\rangle}{\langle \mathsf{now}\, c\, \mathsf{then}\, B1\, \mathsf{else}\, B2, sst\rangle \longrightarrow \langle B1', sst'\rangle} \quad \text{if } sst \vdash^{-} c$$

$$\textbf{R6}\quad \frac{\langle B1, sst\rangle \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, B1\, \mathsf{else}\, B2, sst\rangle \longrightarrow \langle B1, sst\rangle} \quad \text{if } sst \vdash^{-} c$$

$$\textbf{R7}\quad \frac{\langle B2, sst\rangle \longrightarrow \langle B2', sst'\rangle}{\langle \mathsf{now}\, c\, \mathsf{then}\, B1\, \mathsf{else}\, B2, sst\rangle \longrightarrow \langle B2', sst'\rangle} \quad \text{if } sst \not\vdash^{-} c$$

$$\textbf{R8}\quad \frac{\langle B2, sst\rangle \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, B1\, \mathsf{else}\, B2, sst\rangle \longrightarrow \langle B2, sst\rangle} \quad \text{if } sst \not\vdash^{-} c$$

$$\textbf{R9}\quad \frac{\langle B1, sst\rangle \longrightarrow \langle B1', sst'_1\rangle,\ \langle B2, sst\rangle \longrightarrow \langle B2', sst'_2\rangle}{\langle B1 || B2, sst\rangle \longrightarrow \langle B1' || B2', sst'_1 \sqcup sst'_2\rangle}$$

$$\textbf{R10}\quad \frac{\langle B1, sst\rangle \longrightarrow \langle B1', sst'\rangle,\ \langle B2, sst\rangle \not\longrightarrow}{\langle B1 || B2, sst\rangle \longrightarrow \langle B1' || B2, sst'\rangle}$$

$$\textbf{R11}\quad \frac{\langle B1, sst\rangle \not\longrightarrow,\ \langle B2, sst\rangle \longrightarrow \langle B2', sst'\rangle}{\langle B1 || B2, sst\rangle \longrightarrow \langle B1 || B2', sst'\rangle}$$

$$\textbf{R12}\quad \frac{\langle B, sst_1 \cup \exists x sst_2\rangle \longrightarrow \langle B', sst'\rangle}{\langle \exists^{sst_1} x B, sst_2\rangle \longrightarrow \langle \exists^{sst'} x B', sst_2 \sqcup \exists x sst'\rangle}$$

$$\textbf{R13}\quad \langle \mathsf{p}(x), sst\rangle \longrightarrow \langle B, sst\rangle \quad \text{if } \mathsf{p}(x) ::= B \in D$$

Figure 3. Abstract operational semantics of tccp

In the semantic rules, we are assuming that an abstraction operator $\rho : \wp(\wp(\mathcal{C})) \to \wp(\wp(\mathcal{C}))$ has been provided and it has the properties discussed in Section 3.1. We have dropped the subindex $\rho$ from $\vdash^{+}$, $\vdash^{-}$ and $\sqcup$ in order to simplify the presentation. For the same reason, in the sequel, we write $sst \vdash^{+} c$, $sst \vdash^{-} c$ and $sst \sqcup c$ for $sst \vdash^{+} \{\{c\}\}$, $sst \vdash^{-} \{\{c\}\}$ and $sst \sqcup \{\{c\}\}$.

The two main points of the abstract semantics are the new $\mathsf{ask}^{\alpha}$ agent and the abstraction of the conditional agent. We show the transition rules for each agent in Figure 3.[7] A *configuration* of the form $\langle \Gamma, sst\rangle$ represents a computation state, where $\Gamma$ is a multiset of agents and $sst \in \wp(\wp(\mathcal{C}))$ is an abstract store.

Let us now explain the main differences between the abstract semantics and the concrete one defined in [4]. The first important point to observe in the semantics is the use of the two abstract entailment relations $\vdash^{+}$ and $\vdash^{-}$. The

[7] In rule **R12**, the superscript in $\exists^{d} B$ represents the information accumulated during the execution of the agent $B$. See [4] for details.

application of the different rules in Figure 3 depends on these two abstract entailment relations. In particular, under-approximation is used only for the conditional agent. There is a completely new rule (**R4**), which defines the semantics for the instantaneous choice agent ($\mathsf{ask}^{\alpha}$). This rule states that, provided agent $A_j$ can evolve to agent $A'_j$, the instantaneous choice can evolve to $A'_j$. It is important to remark the timing difference between rule **R3** and rule **R4**. Both of them introduce non-determinism but a time unit is consumed in the first one before executing the agent in the body of the ask agent, whereas in the second rule, non-determinism is introduced instantaneously.

## *3.3 Program Abstraction*

In this section, we give a first step towards a source-to-source transformation of tccp programs into abstract programs which represent an approximate model of the system. For each tccp agent $A$, we inductively construct a corresponding abstract tccp agent $\alpha(A)$ as is shown in Figure 4.

**Stop agent.** $\alpha(\mathsf{stop}) = \mathsf{stop}$. **Tell agent.** $\alpha(\mathsf{tell(c)}) = \mathsf{tell}(c)$.

**Choice agent.** $\alpha(\sum_{i=0}^{n} \mathsf{ask}(c_i) \to B_i) = \sum_{i=0}^{n} \mathsf{ask}(c_i) \to \alpha(B_i)$.

**Conditional agent.** $\alpha(\mathsf{now}\, c\, \mathsf{then}\, B1\, \mathsf{else}\, B2) =$

$\mathsf{now}\ c\ \mathsf{then}\ \alpha(B1)\ \mathsf{else}\ \mathsf{ask}^{\alpha}(c) \to \alpha(B1) + \mathsf{ask}^{\alpha}(\mathbf{true}) \to \alpha(B2)$

**Parallel agent.** $\alpha(B1 || B2) = \alpha(B1) || \alpha(B2)$.

**Hiding agent.** $\alpha(\exists x\ B) = \exists x\ \alpha(B)$, where $x$ is a variable.

**Procedure Call agent.** $\alpha(p(x)) = p(x)$ where $x$ is a variable of the constraint system and there exists a procedure declaration as $p(x) ::= B$.

Figure 4. $\alpha$-transformation for tccp programs

The intuitive idea of the transformation of the conditional agent is as follows. We let $\vdash^-$ ($\vdash^+$) represent a suitable under (over) approximation of the entailment relation $\vdash$ of the constraint system. In order to represent the possible conditional execution in the concrete model by an execution in the corresponding abstract model, we consider the following four possible cases, where $st \in \wp(\mathcal{C})$ and $sst \in \wp(\wp(\mathcal{C}))$ are, respectively, the concrete store and the abstract one, and $\rho(\{st\}) \subseteq sst$.

- if $st \vdash c$ and $sst \vdash^- c$, then $B1$ is executed in both the concrete and the abstract models,

- if $st \vdash c$ and $sst \not\vdash^- c$, then agent $B1$ is executed in the concrete model whereas any of the agents $B1$ or $B2$ could be executed in the abstract one,
- if $st \not\vdash c$ but $sst \vdash^+ c$, then in the concrete model agent $B2$ is executed, whereas any of the agents $B1$ or $B2$ could be executed in the abstract one,
- if $st \not\vdash c$ and $sst \not\vdash^+ c$, then both the abstract and the concrete models execute agent $B2$.

Note that the combination of the two abstract entailment relations allows us to precisely approximate the behavior of now for the first case above, whereas this could not be obtained by using only $\vdash^+$.

## 4 Correctness

In this section, we demonstrate that some additional conditions concerning the suspension behavior of the program are needed for the abstract semantics of tccp programs to correctly approximate the standard one. In the case these conditions hold, we develop an abstraction-by-transformation method which allows us to approximate tccp computations. Finally, we show how the abstract semantics can be refined in order to preserve suspension. However, the transformation method for implementing abstraction can not be directly applied to the refined abstract semantics anymore.

### 4.1 Conditions to obtain correctness

Given a tccp program (a process) $P$ of the form $D.\Gamma_0$ and an initial configuration $\langle\Gamma_0, st_0\rangle$, a *trace* $t$ of $P$ starting at $\langle\Gamma_0, st_0\rangle$ is a sequence of configurations $t = \langle\Gamma_0, st_0\rangle \longrightarrow \cdots$ built using the transition rules given by the operational semantics. Let $\mathcal{O}(P)(\langle\Gamma_0, st_0\rangle)$ denote the set of traces generated using the standard operational semantics given in [4]. We say that a concrete trace $t = \langle\Gamma_0, st_0\rangle \longrightarrow \cdots \in \mathcal{O}(P)(\langle\Gamma_0, st_0\rangle)$ is *erroneous* iff $\exists i \geq 0. st_i$ is not consistent.

Similarly, given an abstraction $\rho$, let $\mathcal{A}_\rho(P^\alpha)(\langle\Gamma_0, sst_0\rangle)$ denote the set of abstract traces generated by the abstract program $P^\alpha$ using the abstract operational semantics given by Figure 3. Note that abstract program $P^\alpha$ may include the new agent ask$^\alpha$.

Given a trace $t = \langle\Gamma_0, st_0\rangle \longrightarrow \langle\Gamma_1, st_1\rangle \longrightarrow \cdots \in \mathcal{O}(P)(\langle\Gamma_0, st_0\rangle)$, we denote with $\alpha(t)$ the abstract trace obtained by applying the transformation $\alpha$ presented previously to the agents in the configurations of $t$, and abstracting the corresponding stores using $\rho$, that is, $\alpha(t) = \langle\alpha(\Gamma_0), \rho(\{st_0\})\rangle \longrightarrow \langle\alpha(\Gamma_1), \rho(\{st_1\})\rangle \longrightarrow \cdots$. In addition, given two abstract traces $t_1^\alpha = \langle\Gamma_0^\alpha, sst_{01})\rangle \longrightarrow \langle\Gamma_1^\alpha, sst_{11}\rangle \longrightarrow \cdots$ and $t_2^\alpha = \langle\Gamma_0^\alpha, sst_{02}\})\rangle \longrightarrow \langle\Gamma_1^\alpha, sst_{12}\rangle \longrightarrow \cdots$, we write $t_1^\alpha \sqsubseteq t_2^\alpha$ when $\forall i \geq 0. sst_{i1} \subseteq sst_{i2}$.

**Conditions for Correctness (CC)** Assume that function $\rho$ preserves the local suspension from the concrete configurations to the abstract ones, that is, for all configuration $\Gamma$ and for each store $st$, if $\langle \Gamma, st\rangle \not\longrightarrow$ and $\rho(\{st\}) \subseteq sst$ then $\langle \alpha(\Gamma), sst)\rangle \not\longrightarrow$.

**Theorem 4.1** *Given a* tccp *program $P$, an initial configuration $\langle \Gamma_0, st_0\rangle$ and an abstraction function $\rho : \wp(\wp(\mathcal{C})) \rightarrow \wp(\wp(\mathcal{C}))$ satisfying condition* **CC***, then for each non erroneous trace $t \in \mathcal{O}(P)(\langle \Gamma_0, st_0\rangle)$ there exists an abstract trace $t^\alpha \in \mathcal{A}_\rho(\alpha(P))(\langle \alpha(\Gamma_0), \rho(\{st_0\})\rangle)$ such that $\alpha(t) \sqsubseteq t^\alpha$.*

**Example 4.2** The abstraction given in Example 3.11 of the tccp program illustrated in Figure 1 satisfies **CC**. Note that the concrete model never suspends if stream $C$ has a message, and the same applies to the abstract model. In addition, if $C$ has no message, both the concrete and the abstract model suspend. Therefore, Theorem 4.1 may be applied in this case. This abstraction is useful to check properties like "the photocopier is switched off when it is inactive during $Max$ time units".

The abstraction may modify some time aspects, in such a way that abstract agents are not correctly synchronized, as illustrated by the following example.

**Example 4.3** Consider the abstraction given by Figure 2 which takes into account the divisibility by four of variable $X$ (recall that the abstract store $\mathtt{n\,mod\,4} = \mathtt{0}$ represents the stores where $X = n$ and $n$ satisfies this condition). Figure 5 shows that if **CC** does not hold, the abstract model does not correctly simulate the original one. In the figure, the new agents A' and B' appear since A and B could evolve to these agents when rules **R5** or **R7** are applied.

## *4.2 Implementation of the abstract semantics*

In Section 3 we showed how it is possible to abstract tccp programs. An abstract semantics for the abstract model and a program transformation of the program was formulated. In this section, we show how we can implement the abstract semantics by using only the original tccp agents, i.e., eliminating the $\mathsf{ask}^\alpha$ agent thus obtaining a source-to-source trasnformation from the concrete to the abstract model. We also prove that the implementation corresponds to the intended behavior of the abstract semantics.

We first recall the transformation of the conditional agent:

$$\alpha(\mathsf{now}\, c\, \mathsf{then}\, B1\, \mathsf{else}\, B2) = \mathsf{now}\, c\, \mathsf{then}\, \alpha(B1)\, \mathsf{else}\, \mathsf{ask}^\alpha(c) \rightarrow \alpha(B1) + \mathsf{ask}^\alpha(\mathbf{true}) \rightarrow \alpha(B2)$$

If we simply substitute the $\mathsf{ask}^\alpha$ agent by the original $\mathsf{ask}$ one, then the body agent in the abstract model is executed in the current time instant, whereas in the new version it is executed in the next one. This means that

Concrete Trace

| STORE | AGENTS |
|---|---|
| X=0 | ask(X=4) → tell(Y=2) \|\| ask(true) → ask(true) → now Y=2 then A else B |
| X=0 | ask(X=4) → tell(Y=2) \|\| ask(true) → now Y=2 then A else B |
| X=0 | ask(X=4) → tell(Y=2) \|\| now Y=2 then A else B |
| X=0 | ask(X=4) → tell(Y=2) \|\| B' |

Abstract trace

| STORE | AGENTs |
|---|---|
| `n mod 4 = 0` | ask(X=4) → tell(Y=2) \|\| ask(true) → ask(true) → now Y=2 then A else B ask(true) → ask(true) → now Y=2 then A else B |
| `n mod 4 = 0` | tell(Y=2) \|\| ask(true) → now Y=2 then A else B |
| `n mod 4 = 0` ∧ Y=2 | now Y=2 then A else B |
| `n mod 4 = 0` ∧ Y=2 | A' |

Figure 5. A non correct abstract model

other agents that could eventually be executed concurrently could introduce information in the store which interferes with the execution, thus changing the program semantics.

In order to overcome this problem, we strategically introduce delays in the program. For this purpose, we first apply a static analysis to the program which determines what is the maximum number ($N$) of nested conditional agents in the program, and associates to each occurrence of timing agents an integer number which represents its relative depth (with respect to nested conditional agents). $A_\epsilon$ denotes that the relative depth of $A$ is $\epsilon$.

In short, the transformed program is obtained by introducing $N-\epsilon$ delays for each $A_\epsilon$ agent in the program. Note that it is also necessary to introduce the corresponding delay in the case when the choice agent suspends. This implies a more elaborated transformation which we do not present in this work due to the limited space. In Figure 6 we show the transformation associated to each agent. Notation $(\mathsf{ask}(\mathbf{true}) \rightarrow)^{N-\epsilon}$ means that the $\mathsf{ask}(\mathbf{true})$ agent is replicated $N-\epsilon$ times.

Intuitively, we can imagine that we split each time instant into $N$ parts. When we abstract a concrete program, the structure of the abstracted program ensures that all agents corresponding to concrete agents (except for the conditional agent) are executed in the last part of each $N$ block. Furthermore, the execution of conditional agents will start during the corresponding part of the block depending on its relative depth.

Given a program $P$, we call $P^\beta$ to the $\beta$-program transformation. Next we prove that the $\beta$-transformation described above is correct. We say that

**Stop agent.** $\beta(\mathsf{stop}_\epsilon) = (\mathsf{ask}(\mathbf{true}) \rightarrow)^{N-\epsilon}\ \mathsf{stop}$.

**Tell agent.** $\beta(\mathsf{tell(c)}_\epsilon) = (\mathsf{ask}(\mathbf{true}) \rightarrow)^{N-\epsilon}\ \mathsf{tell}(c)$.

**Choice agent.**
$\beta(\sum_{i=0}^{n} \mathsf{ask}(c_i)_\epsilon \rightarrow B_i) = (\mathsf{ask}(\mathbf{true}) \rightarrow)^{N-\epsilon} \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow \beta(B_i)$.

**Conditional agent.** $\beta(\mathsf{now}\, c\, \mathsf{then}\, B1\, \mathsf{else}\, B2) =$

$\mathsf{now}\ c\ \mathsf{then}\ \beta(B1)\ \mathsf{else}\ (\mathsf{ask}(c) \rightarrow \beta(B1) + \mathsf{ask}(\mathbf{true}) \rightarrow \beta(B2))$

**Parallel agent.** $\beta(B1||B2) = (\beta(B1)||\beta(B2))$.

**Hiding agent.** $\beta(\exists x\ B) = \exists x\ \beta(B)$, where $x$ is a variable.

**Procedure Call agent.** $\beta(p(x)_\epsilon) = (\mathsf{ask}(\mathbf{true}) \rightarrow)^{N-\epsilon}\ p(x)$ where $x$ is a variable of the constraint system and there exists a procedure declaration as $p(x) ::= B$.

Figure 6. $\beta$-transformation of tccp programs

the transformation is correct if, for all $P$, $P^\beta$ simulates the behavior of $P^\alpha$ obtained by applying the $\alpha$-transformation described in Section 3.

We define the notion of observable for $\beta$-programs as $NOb(P^\beta)(\langle\Gamma_0, sst_0\rangle) = \{\langle\Gamma_{0*N}, sst_{0*N}\rangle \longrightarrow \langle\Gamma_{1*N}, sst_{1*N}\rangle \longrightarrow \langle\Gamma_{2*N}, sst_{2*N}\rangle \longrightarrow \ldots \mid \langle\Gamma_0, sst_0\rangle \longrightarrow \langle\Gamma_1, sst_1\rangle \longrightarrow \langle\Gamma_2, sst_2\rangle \ldots \in \mathcal{A}_\rho(P^\beta)(\langle\Gamma_0, sst_0\rangle)\}$

**Theorem 4.4** *Given a* tccp *program $P$, an initial configuration $\langle\Gamma_0, sst_0\rangle$ and an abstraction function $\rho : \wp(\wp(\mathcal{C})) \rightarrow \wp(\wp(\mathcal{C}))$ then*

$$NOb(P^\beta)(\langle\Gamma_0, sst_0\rangle) = \mathcal{A}_\rho(P^\alpha)(\langle\Gamma_0, sst_0\rangle)$$

The proof is based on the demonstration that the $\beta$-abstraction models the $\alpha$-abstraction (and vice-versa), and is done by induction on both the length of traces (for the procedure call case) and the structure of agents.

## 4.3 A Correct Abstract Semantics

We finalize the discussion about the correct simulation of a tccp program by giving an abstract semantics where the problem shown by Figure 5 is solved. The idea is to non-deterministically allow the repetition of an abstract configuration when it is possible that some concretization may suspend. Note that again, in this case, the use of the abstract entailment relation $\vdash^-$ makes it possible to precisely represent this situation.

Consider the transition system obtained by modifying the abstract semantics given in Figure 3 with the new rules presented by Figure 7, as follows. Rule **R1** is substituted by rule **R1'**, rule **R3'** is added, and rules **R6**, **R8**, **R10** and **R11** are dropped. Note that a configuration containing an ask is replicated with the new abstract semantics when it may suspend in the concrete semantics. This is how to simulate suspension in the abstract semantics.

**R1'** $$\langle \mathsf{stop}, d\rangle \longrightarrow \langle \mathsf{stop}, d\rangle$$

**R3'** $$\langle \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow A_i, d\rangle \longrightarrow \langle \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow A_i, d\rangle \quad \text{if } \forall j.0 \leq j \leq n, d \not\vdash^{-} c_j$$

Figure 7. New rules for a correct abstract semantics of tccp

The new semantics ($\mathcal{A}'_\rho$) gives us the desired result about correctness.

**Theorem 4.5** *Given a* tccp *program $P$ of the form $D.\Gamma_0$, an initial configuration $\langle \Gamma_0, st_0\rangle$ and an abstraction function $\rho : \wp(\wp(\mathcal{C})) \rightarrow \wp(\wp(\mathcal{C}))$, then for each non erroneous trace $t \in \mathcal{O}(P)(\langle \Gamma_0, st_0\rangle)$ there exists an abstract trace $t^\alpha \in \mathcal{A}'_\rho(\alpha(P))(\langle \alpha(\Gamma_0), \rho(\{st_0\})\rangle)$ such that $\alpha(t) \sqsubseteq t^\alpha$.*

**Example 4.6** For instance, consider the traces given in Figure 5, with the new semantics we obtain the abstract simulation shown in Figure 8

New Abstract trace

| STORE | AGENTs |
|---|---|
| `X mod 4 = 0` | ask(X=4) → tell(Y=2) \|\| ask(true) → ask(true) → now Y=2 then A else B |
| `X mod 4 = 0` | ask(X=4) → tell(Y=2) \|\| ask(true) → now Y=2 then A else B |
| `X mod 4 = 0` | ask(X=4) → tell(Y=2) \|\| now Y=2 then A else B |
| `X mod 4 = 0` | ask(X=4) → tell(Y=2) \|\| B' |

Figure 8. A correct abstract model

## 5 Conclusions and Future Work

In this work, we have proposed a methodology which mitigates the state explosion problem in tccp model checking. We have defined a source-to-source transformation for tccp programs which is based on over- and under-approximating constraint systems. The abstraction of the conditional tccp agent introduces some specific difficulties which have been solved by using under-approximation in the abstract semantics; this idea is novel since only over-approximations are typically used when approximating models in the data abstraction approach. The inspiration to combine over- and under-approximation in model–checking tccp comes from [13]. Moreover, we have proved that correctness of the semantics is not guaranteed unless several conditions concerning the suspension of agents do hold. We also provide a simple refinement of the abstract semantics which overcomes this problem. Unfortunately, we obtain a computation model which is not the one of tccp anymore. As future work, we plan to complete our methodology by defining an approximation technique for the properties that must be verified. We intend to formulate this method also as a source-to-source transformation, which will allow us to compare the state-spaces generated by the exhaustive classical method to the one proposed here.

## References

[1] Alpuente, M., Falaschi, M. and Villanueva, A., *Symbolic Model Checking for Timed Concurrent Constraint Programs*, in: *Proc. of III Jornadas de Programación y Lenguajes*, Alicante, 2003.

[2] Ball, T., Podelski, A. and Rajamani, S.K., *Relative completeness of abstraction refinement for software model checking*, in: *TACAS'02*, LNCS **2280**:158–172, (2002).

[3] Ball, T. and Rajamani, S.K., *The slam project: Debugging system software via static analysis*, in: *Proc. of POPL 2002*, 2002, pp. 1–3.

[4] Boer, F. de, Gabbrielli, M. and Meo, M.C., *A Timed Concurrent Constraint Language.*, Information and Computation **161** (2000), pp. 45–83.

[5] Boer, F. de, Gabbrielli, M. and Meo, MC., *A Temporal Logic for reasoning about Timed Concurrent Constraint Programs*, in: *Proc. of 8th Int. Symposium on Temporal Representation and Reasoning* (2001), pp. 227–233.

[6] Clarke, E.M., Emerson, E.A. and Sistla, A.P., *Automatic verification of finite-state concurrent systems using temporal logic specifications*, ACM Transactions on Programming Languages and Systems **8** (1986), pp. 244–263.

[7] Clarke, E.M., Grumberg, O. and Long, D.E., *Model Checking and Abstraction*, ACM Transactions on Programming Languages and Systems **16** (1994), pp. 1512–1542.

[8] Cousot, P. and Cousot, R., *Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints*, in: *Proc. of POPL* (1977), pp. 238–252.

[9] Cousot, P. and Cousot, R., *Systematic Design of Program Analysis Frameworks*, in: *Proc. of POPL* (1979), pp. 269–282.

[10] Dams, D., Gerth, R. and Grumberg, O., *Abstract interpretation of reactive systems*, ACM Transactions on Programming Languages and Systems **19** (1997), pp. 253–291.

[11] Falaschi, M., Policriti, A. and Villanueva, A., *Modeling Timed Concurrent systems in a Temporal Concurrent Constraint language - I, Selected papers from 2000 Joint Conference on Declarative Programming*, ENTCS **48** (2000).

[12] Falaschi, M. and Villanueva, A., *Automatic verification of timed concurrent constraint programs* (2003), submitted for publication.

[13] Gallardo, M., Martínez, J., Merino, P. and Pimentel, E., *$\alpha$spin: a tool for abstract model checking*, Int. Journal on Software Tool for Technology Transfer **online version** http://sttt.cs.uni-dortmund.de/ (2003).

[14] Gallardo, M., Merino, P. and Pimentel, E. *A generalized semantics of promela for abstract model checking*, Formal Aspects of Computing **to appear** (2004).

[15] Giacobazzi, R., Debray, S.K. and Levi, G., *Generalized semantics and abstract interpretation for constraint logic programs*, J. of Logic Progr. **25** (1995), pp. 191–247.

[16] Hatcliff, J., Dwyer, M., Pasareanu, C. and Robby, *Foundations of the bandera abstraction tools*, in: *The Essence of Computation*, LNCS **2566**, pp. 172–203.

[17] Loiseaux, C., Graf, S., Sifakis, J. and Boujjani, S.B.A., *Property preserving abstractions for the verification of concurrent systems*, Formal Methods in System Design **6** (1995), pp. 1–35.

[18] McMillan, K.L., *Symbolic Model Checking: An Approach to the State Explosion Problem* Kluwer Academic, 1993.

[19] Saraswat, V.A., *Concurrent Constraint Programming Languages*, The MIT Press, Cambridge, MA, 1993.

[20] Saraswat, V.A., Jagadeesan, R. and Gupta, V., *Foundations of Timed Concurrent Constraint Programming*, in: *Proc. 9th IEEE Symposium on LICS* (1994), pp. 71–80.