TLP **6** (3): 265–300, 2006. © 2006 Cambridge University Press
doi:10.1017/S1471068406002675 Printed in the United Kingdom

*Automatic verification of timed concurrent constraint programs**

MORENO FALASCHI

Dip. Matematica e Informatica, University of Udine,
Via delle Scienze, 206. I-33100 Udine, Italy
(*e-mail:* falaschi@dimi.uniud.it)

ALICIA VILLANUEVA

Dep. Sistemas Informáticos y Computación, Technical University of Valencia
Camino de Vera s/n. E-46022 Valencia, Spain
(*e-mail:* villanue@dsic.upv.es)

submitted 29 December 2003; revised 7 April 2004; accepted 10 May 2005

Abstract

The language *Timed Concurrent Constraint* (tccp) is the extension over time of the *Concurrent Constraint Programming* (cc) paradigm that allows us to specify concurrent systems where timing is critical, for example *reactive systems*. Systems which may have an infinite number of states can be specified in tccp. *Model checking* is a technique which is able to verify finite-state systems with a huge number of states in an automatic way. In the last years several studies have investigated how to extend model checking techniques to systems with an infinite number of states. In this paper we propose an approach which exploits the computation model of tccp. Constraint based computations allow us to define a methodology for applying a model checking algorithm to (a class of) infinite-state systems. We extend the classical algorithm of model checking for LTL to a specific logic defined for the verification of tccp and to the tccp Structure which we define in this work for modeling the program behavior. We define a restriction on the time in order to get a finite model and then we develop some illustrative examples. To the best of our knowledge this is the first approach that defines a model checking methodology for tccp.

KEYWORDS: Automatic verification, reactive systems, timed concurrent constraint programming, model checking

1 Introduction

Model checking is a technique for formal verification that was defined for finite-state systems. It was first introduced in Clarke and Emerson (1981) and Quielle and Sifakis

* This work has been partially supported by the EU (FEDER) and the Spanish MEC, under grant TIN 2004-7943-C04-02, by ICT for EU-India Cross Cultural Dissemination Project under grant ALA/95/23/2003/077-054, and by the Italian project Cofin'04 AIDA.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

(1982) for verifying *automatically* if a system satisfies a given property. Concurrent systems can be very complicated, and the process of modeling and verifying them by hand can be hard. Thus, the development of formal and fully automatic methods such as model checking is essential. Basically, this technique consists in an exhaustive analysis of the state-space of the system. This exhaustive analysis implies that, in principle, we can apply it only to finite-state systems limiting a lot its applicability. Furthermore, the state-explosion problem is the main drawback even for finite-state systems and for this reason many approaches in the literature try to mitigate it. Two of the main solutions for the state-explosion problem that have been presented in the last years are the symbolic approach (McMillan 1993) and the algorithms for abstract model checking (Dams 1996). The idea which is shared by these approaches is to reduce the number of states of the system.

The different approaches to the model checking problem for infinite state systems can be classified in two categories. The first one corresponds to those approaches that construct an abstract finite model of the system which can be automatically verified (Clarke *et al.* 1994; Loiseaux *et al.* 1995). The second category contains those approaches based on the symbolic reachability analysis where a finite representation of the set of reachable configurations of the system is calculated (Alur *et al.* 1995; Cousot and Halbwachs 1978; Bouajjani *et al.* 1997; Boigelot and Godefroid 1996). The methodologies that make use of regular languages and regular relations are considered in the so-called *regular model checking* approach (Pnueli and Shahar 2000; Kesten *et al.* 1997; Bouajjani *et al.* 2000). Moreover, in Abdulla *et al.* (1999) the notion of abstraction and the notion of symbolic reachability are combined to define a method to verify infinite-state systems. Our approach is novel and makes use of a notion of abstraction based on constraints and a time interval. The notion of constraints is used to collapse the number of states.

In Manna Pnueli (1995), *reactive systems* are defined as those systems that keep exchanging information with their environment at run time. Reactive systems are typically defined as a set of processes working in parallel, hence the family of reactive systems is strictly related to the notion of *concurrency*. In some cases it is not expected that the system terminates but it may continue its execution indefinitely. Examples of such systems are operating systems, communication protocols or some kind of embedded systems. Thus it is quite useful to have a specification language that supports concurrency which makes easier for the user to describe systems. Usually, in model checking, by exploiting concurrency we model the whole system, including the environment. For example, users are represented as a concurrent process which models the possible actions that users can perform to interact with the system.

The language *Temporal Concurrent Constraint Programming* (tccp) extends the *Concurrent Constraint Programming* (cc) paradigm defined in Saraswat (1989) with a notion of time. This extension is suitable for modeling reactive systems. Actually, in the literature you can find two similar languages which extend cc with some notion of time: the tcc language first presented in Saraswat *et al.* (1994) and the ntcc language defined in Nielsen *et al.* (2002). tccp is a declarative language defined in Boer *et al.* (2000) that handles constraints which is a key characteristic for the results which we achieve in the present work. Our idea is to take advantage of

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

the natural properties of the language to define a model-checking algorithm that allows us to verify reactive systems specified in tccp. Note that when we speak of reactive systems we are not limiting ourselves to finite-state systems. The tccp language allows us to model infinite-state systems, hence we tackle the problem of model checking for infinite-state systems. We show how the constraint nature of the language and the fact that it has a built-in notion of time can be exploited usefully.

Some related works can be found in the literature where constraints are used for solving similar problems. In Delzanno and Podelski (1999, 2001), the authors present a method that allows them to verify a communication protocol with an infinite number of states in the sense that they prove that a client-server protocol is correct for an arbitrary number of processes (clients). This could not be proved by using classical approaches to model checking, however it become possible thanks to the use of the notion of constraint.

The model-checking technique can be divided into three main phases; specification, modeling and verification. In this work, we use the notion of constraint in the three phases of the model-checking technique. First, we introduce the notion of constraint in the constructed model of the system. We note that constraints are able to represent in a compact manner a set of possible values that the system variables can take (i.e., a possibly infinite set of states if we use the classical notion of state). In the second phase we use a logic able to handle constraints for specifying the property to be verified. Such logic was presented in Boer *et al.* (2001) and revisited in Boer *et al.* (2002). The last phase of the model-checking technique consists in defining an algorithm that determines whether the system satisfies the property by using the two outputs of the previous phases. In this work we extend the classical algorithm defined for LTL to the constrained approach. Note that we can take as a reference the classical algorithm because we use a logic able to handle constraints, and this makes possible to combine it with the tccp Structure defined in this paper to model the system. Since this structure contains constraints, it would not be possible to use a classical temporal logic directly. To the best of our knowledge this is the first time that a model-checking algorithm for systems specified with the tccp language is defined. Some of the results in this work have been included in Villanueva's doctoral thesis (Villanueva 2003).

In Falaschi *et al.* (2000a, 2000b) we presented a framework that allowed us to build a graph structure as a first step for applying the model-checking technique to tcc programs. tcc is a language similar to tccp for programming embedded systems. The main differences between tcc and the language that we consider here is in the deterministic nature of the tcc language versus the non-determinism, and the *monotonicity* of the store in tccp. Monotonicity means that the store of the system always increases. tcc is not monotonic since the store is reset when passing from one time instant to the following one. These differences make the graph structures defined in Falaschi *et al.* (2000a, 2000b) and in this work completely different. We will show these differences in detail in the following sections. Moreover, only the modeling process of the method was presented in Falaschi *et al.* (2000a, 2000b), whereas in this paper we provide the logic used for the specification of the property and the model-checking algorithm as well.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

This paper is organized as follows. In Section 2 we introduce some basic theoretic notions. In Section 3 we present the basic notions of the tccp language. Then, in Section 4 we describe the method to construct an adequate model of the system, which is shown to model correctly the language operational semantics. In Section 5 we present the logic for specifying the properties of our system. In Section 6 we define the algorithm that applies the model-checking technique to this model and show its correctness. Section 7 discusses some related work. Finally, in Section 8 final remarks and future work are discussed.

2 Preliminaries

In this section we present some definitions necessary to follow the technical details of this work. For a quick reading it is possible to skip to Section 3.

A *Constraint System* is a system of partial information. We follow the definition of Saraswat *et al.*:

Definition 1 (*Simple constraint system (Saraswat* et al. *1991)*)
Let D be a non-empty set of *tokens* or *primitive constraints*. A *simple constraint system* is a structure $\langle C, \vdash \rangle$ where $\vdash \subseteq \wp_f(C) \times C$ is an *entailment relation* satisfying:

C1 $u \vdash P$ whenever $P \in u$,
C2 $u \vdash Q$ whenever $u \vdash P$ for all $P \in v$ and $v \vdash Q$.

Moreover, an element of $\wp_f(C)$ is called a *finite constraint* and $\vdash$ is extended to $\wp_f(C) \times \wp_f(C)$ in the obvious way. Finally, $u \approx v$ iff $u \vdash v$ and $v \vdash u$. We also say that $u \geqslant v$ when $v \vdash u$.

Definition 2 (*Cylindric constraint system (Saraswat* et al. *1991)*)
We define a cylindric constraint system as a structure $\langle C, \vdash, \mathscr{V}, \{\exists_x \mid x \in \mathscr{V}\} \rangle$ such that $\langle C, \vdash \rangle$ is a simple constraint system, $\mathscr{V}$ is an infinite set of *variables* and, for each $x \in \mathscr{V}$, $\exists_x : \wp_f(C) \to \wp_f(C)$ is an operation satisfying:

E1 $u \vdash \exists_x u$,
E2 $u \vdash v$ implies $\exists_x u \vdash \exists_x v$,
E3 $\exists_x(u \sqcup \exists_x v) \approx \exists_x u \sqcup \exists_x v$,
E4 $\exists_x \exists_y u \approx \exists_y \exists_x u$.

$\exists_x$ is called the *existential quantifier* or *cylindrification operator*.

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

D1 $\emptyset \vdash \delta_{xx}$,
D2 if $y \neq x, z$ then $\{\delta_{xz}\} \approx \exists_y \{\delta_{xy}, \delta_{yz}\}$,
D3 if $x \neq y$ then $\{\delta_{xy}\} \sqcup \exists_x(u \sqcup \{\delta_{xy}\}) \vdash u$.

We define an *element* c of a cylindric constraint system $\langle C, \vdash \rangle$ as a subset of C closed by entailment, i.e., such that $u \subseteq_f c$ and $u \vdash P$ implies $P \in c$.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

3 Timed concurrent constraint language

The tccp language was developed in Boer *et al.* (2000). It was designed as a computational model which allows one to model reactive and real-time systems. Thus, it is possible to specify and to verify distributed, concurrent systems where the notion of time is a crucial question. tccp is based on the cc paradigm (Saraswat 1989; Saraswat and Rinard 1990; Saraswat *et al.* 1991) that was presented as a general concurrent computational model.

The computational model of cc is defined by means of a global store and a set of defined agents that can add (tell) information into the store or check (ask) whether a constraint is entailed by the store. Computations evolve as an accumulation of information into a global store. In tccp the agents defined for cc are inherited. The model is enriched with a new agent and a *discrete global clock*. It is assumed that ask and tell actions take one time-unit and the parallel operator is interpreted in terms of maximal parallelism. Computation evolves in steps of one time-unit. It is assumed that the response time of the constraint solver is constant, independently of the size of the store. In practice some restrictions (mentioned below) are taken to ensure that these hypothesis are reasonable (the reader can see Boer *et al.* (2000) for details).

To model reactive systems it is necessary to have the ability to describe notions as *timeout* or *preemption*. The timeout behavior can be defined as the ability to wait for a specific signal and, if a limit of time is reached and such signal is not present, then an exception program is executed. The notion of preemption is the ability to abort a process when a specific signal is detected. In tccp these behaviors can be modeled by using the new conditional agent (not present in cc)

$$\mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B$$

which tests if in the current time instant, the store entails the constraint c and if it occurs, then in the same time instant it executes the agent A; otherwise, it executes B (in the same time instant). A limit for the number of nested conditional agents is imposed in order to ensure the bounded time response of the constraint solver within a time instant.

3.1 Syntax

The tccp language is parametric to an underlying cylindric constraint system as defined in Section 2. Since now we assume that $\mathscr{C} = \langle C, \vdash, \mathscr{V}, \exists \rangle$ is the underlying constraint system for tccp. Given $\mathscr{C}$, in Figure 1 we show the syntax of the agents of the language. We assume that c and c_i are finite constraints (i.e. elements) in $\mathscr{C}$.

The Parallel and Hiding agents are inherited from the cc model and behave in the same way. Thus, the Parallel agent represents concurrency, whereas the Hiding operator makes a variable local to some process. Also the Tell, Choice and Procedure Call agents were present in the cc model, but in tccp they have a different semantics since in the timed model, these three agents cause extension over time. The Tell agent adds the information c to the store, but this information is available to other agents only in the following time instant. Therefore, we can say that the tell action takes

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

(Agents)	A ::= $\mathsf{tell}(c)$	– Tell
	| stop	– Stop
	| $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$	– Choice
	| $\mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, A$	– Conditional
	| $A \parallel A$	– Parallel
	| $\exists x\ A$	– Hiding
	| $\mathsf{p}(x)$	– Procedure Call
(Declarations)	D ::= $D.D$	
	| $\mathsf{p}(x)$:- A	
(Program)	P ::= $D.A$	

Fig. 1. tccp syntax (following F. de Boer *et al.*).

one unit of time. The same thing occurs with the Choice and Procedure Call agents. Thus, when we execute the $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ agent, the execution of A_i starts in the next time instant. Note that the Choice agent models the nondeterministic behavior of the language, thus nondeterminism is always associated to a time delay.

Finally, the Conditional agent ($\mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B$) is the new agent introduced in the model in order to capture negative information. It behaves within a time unit in the sense that the condition is checked *in the same instant of time* as the execution of the corresponding agent is started. In particular, if the guard is satisfied, then A will be executed, otherwise the agent B will be executed (we note that B is executed also in the case when the store entails neither c nor $\neg c$). If we have two nested conditional agents, then the guards are recursively checked within the same time instant. This is the reason why tccp needs a restriction about the maximum number of nested conditional agents.

3.2 tccp *Operational semantics*

In Figure 2 it is shown the operational semantics for tccp as described in Boer *et al.* (2000). Each transition step takes one unit of time. In a configuration (**Conf**) there are two components: a set of agents and a finite constraint representing the store. The transition relation $\longrightarrow \subseteq \mathbf{Conf} \times \mathbf{Conf}$ is the least relation that satisfies the rules in Figure 2. We can say that the transition relation characterizes the (temporal) evolution of the system.

Since tccp interprets concurrency in terms of *maximal parallelism*, we assume that there are as many processors as needed to execute a program. This behavior is described by means of rules **R7**, **R8** and **R9** where the reader can see that whenever it is possible, two agents are executed concurrently.

Rules **R3**, **R4**, **R5** and **R6** describe the operational semantics for the conditional agent. Note that the different possible behaviors depend on the store and on the initial configuration. Rule **R10** shows the semantics for the Hiding operator. Intuitively, the rule says that, if there exists a transition $\langle A, d \sqcup \exists_x c\rangle \longrightarrow \langle B, d'\rangle$, then d' is the local information produced by A; moreover, this local information d' must be hidden from the main process.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

R1	$\langle \mathsf{tell}(c), d\rangle \longrightarrow \langle \mathsf{stop}, c \sqcup d\rangle$	
R2	$\langle \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i, d\rangle \longrightarrow \langle A_j, d\rangle$	$j \in [1,n]$ and $d \vdash c_j$
R3	$\dfrac{\langle A,d\rangle \longrightarrow \langle A',d'\rangle}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,d\rangle \longrightarrow \langle A',d'\rangle}$	$d \vdash c$
R4	$\dfrac{\langle A,d\rangle \not\longrightarrow}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,d\rangle \longrightarrow \langle A,d\rangle}$	$d \vdash c$
R5	$\dfrac{\langle B,d\rangle \longrightarrow \langle B',d'\rangle}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,d\rangle \longrightarrow \langle B',d'\rangle}$	$d \nvdash c$
R6	$\dfrac{\langle B,d\rangle \not\longrightarrow}{\langle \mathsf{now}\ c\ \mathsf{then}\ A\ \mathsf{else}\ B,d\rangle \longrightarrow \langle B,d\rangle}$	$d \nvdash c$
R7	$\dfrac{\langle A,c\rangle \longrightarrow \langle A',c'\rangle \quad \langle B,c\rangle \longrightarrow \langle B',d'\rangle}{\langle A \| B,c\rangle \longrightarrow \langle A' \| B', c' \sqcup d'\rangle}$	
R8	$\dfrac{\langle A,c\rangle \longrightarrow \langle A',c'\rangle \quad \langle B,c\rangle \not\longrightarrow}{\langle A \| B,c\rangle \longrightarrow \langle A' \| B,c'\rangle}$	
R9	$\dfrac{\langle A,c\rangle \longrightarrow \langle A',c'\rangle \quad \langle B,c\rangle \not\longrightarrow}{\langle B \| A,c\rangle \longrightarrow \langle B \| A',c'\rangle}$	
R10	$\dfrac{\langle A, d \sqcup \exists_x c\rangle \longrightarrow \langle B,d'\rangle}{\langle \exists^d x A,c\rangle \longrightarrow \langle \exists^{d'} x B, c \sqcup \exists_x d'\rangle}$	
R11	$\langle \mathsf{p}(x), c\rangle \longrightarrow \langle A, c\rangle$	$\mathsf{p}(x) :- A \in D$

Fig. 2. Operational semantics for tccp language extracted from F. de Boer *et al.*

The observable behavior of the language is defined from the transition system described in Figure 2 and considers the input/output of finite and infinite computations:

Definition 3 (*Observable*)
Let A be an agent from the tccp language, the operational behavior is given by the set of resulting stores computed by A for each given input store, considering finite and infinite computations.

$$\mathcal{O}(A) = \{d \mid \langle A, c\rangle \longrightarrow \ldots \langle B, c_i\rangle \longrightarrow \ldots, \text{ where } d \equiv \{c, c_1, \cdots, c_i, \cdots\}\}$$

3.3 Practical example

We can find in the literature a variety of examples of systems that can be modeled using the tccp language. Here we develop a typical system: a microwave oven. In Figure 3 the reader can see the behavior of a microwave. We can note that, for example, if we are in a state where the door of the microwave is closed, the system

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

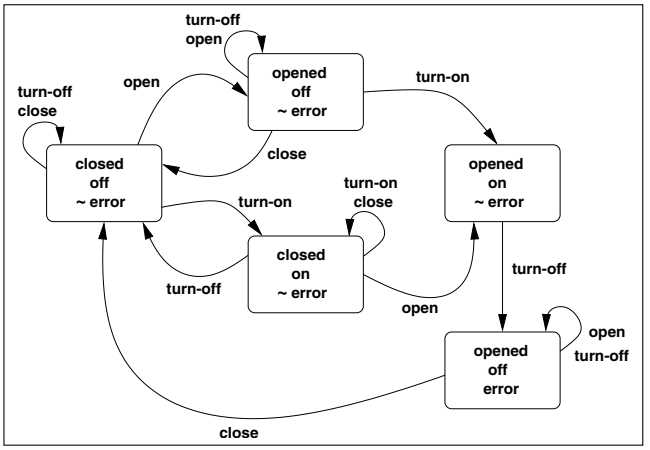


Fig. 3. Example: the microwave system.

```
microwave_error(Door, Button, Error) :-
    ∃ D, B, E( tell(Error = [_|E]) || (tell(Door = [_|D]) || (tell(Button = [_|B]) ||
        (now (Door = [open | D] ∧ Button = [on | B]) then
            (∃ E 1(tell(E = [yes | E 1]))||
            ∃ B 1(tell(B = [off | B 1])))
        else
            ∃ E 1(tell(E = [no | E 1]))||
        microwave_error(D, B, E)))).
```

Fig. 4. Example of a tccp program: a simple error controller.

is turned-off and no error is detected. If, from that state, we open the door, then we move to the state on the top of the figure.

The whole system example is inspired in the system for a microwave control shown in the classical literature (Clarke *et al.* 1999). However, we have considered only a subpart of the system to easily use this example as a reference in this work. In Figure 4 we show the tccp program which models a reduced part of the microwave system. In particular, it models the part of the system which detects if the door is open when the microwave is turned-on.

Looking into the program code, we can observe that a Conditional agent checks if the door is open when the microwave is turned-on. In that case, it forces (with the Tell agent) that in the following time instant, the microwave is turned-off and an error signal is emitted. If it is not true that the door is open and the microwave is working on, then the program simply emits (via the Tell agent) a signal of *no* error that will be available in the global store in the following time instant. Therefore, this example corresponds to the part of the system which avoids wrong behaviors such as those in Figure 3 which are represented by the two states on the right.

This simple example allows us to illustrate the fact that tccp is not able to model strong preemption, i.e., it is not possible to turn-off the start button in the same instant when the error is detected. Actually, it is possible to start the execution of

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

the agent that turns the button off, but the fact that it has been turned off is visible only in the following time instant.

4 tccp model checking

The cc paradigm has some interesting features which allow us to define a model-checking algorithm for reactive systems. We define a model-checking algorithm which uses a time interval (provided by the user) to restrict the state-space of the system in the cases when the algorithm does not terminates. The fact that the time is in the semantics makes reasonable the use of such restriction since the user knows how much time is needed to have a response from the system. The reader could think that the restriction to an interval of time could make the algorithm incomplete in too many cases. In the following sections we show that the time interval is not always used. Obviously, the user must provide a reasonable time interval. Moreover, if the limit is reached and the verification is terminated, then we obtain an over-approximation of the system thus some properties can still be checked. The idea to limit the verification to a time limit is not new. It has been used in different approaches, for example in Alur *et al.* (1997).

Let us now develop a model-checking technique to tccp programs. The key ideas are that we use the notion of constraint which is underlying the language to have a compact model of the system first, and second, to handle the model directly to verify properties.

In the following we describe in detail the three main phases which implement the model-checking algorithm. We also illustrate each phase with the application of the method to the microwave example.

4.1 Model construction

The first task of the method corresponds to the construction of the model of the system. In classical approaches, *Kripke Structures*[1] are used to model the system behavior; in our approach we define a similar structure called tccp Structure whose states are essentially a conjunction of constraints of the underlying constraint system. The idea is to automatize the construction of the model of the system from the specification. In other words, we take a program written in tccp, and a model of the system behavior is constructed in an automatic way.

4.1.1 Program labeling

First of all, we need a labeled version of the specification in order to construct the model of the system automatically. We adapt the idea introduced in Manna and Pnueli (1995) to our framework: a different label is assigned to each occurrence of an agent. Labels allow us to identify during the model construction in which point of

[1] Kripke Structures were defined in Hughes and Creswell (1968). The definition can also be seen in Clarke *et al.* (1999), for example.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

$$
\begin{array}{l}
\{l_{\mathrm{p}_0}\}\ \mathsf{microwave_error}(\mathrm{Door}, \mathrm{Button}, \mathrm{Error}) : - \\
\quad \{l_{\mathrm{e}_1}\}\ \exists D, B, E\,(\{l_{||_2}(\{l_{t_3}\}\mathsf{tell}(\mathrm{Error} = [_|E])||\ (\{l_{||_4}(\{l_{t_5}\}\ \mathsf{tell}(\mathrm{Door} = [_|D])\ || \\
\qquad \{l_{||_6}\}(\{l_{t_7}\}\mathsf{tell}(\mathrm{Button} = [_|B])\ || \\
\quad \{l_{||_8}\}\,(\{l_{\mathrm{now}_9}\}\ \mathsf{now}\,(\mathrm{Door} = [\mathrm{open} \mid D] \wedge \mathrm{Button} = [\mathrm{on} \mid B])\,\mathsf{then} \\
\qquad \{l_{||_{10}}\}\,(\{l_{\mathrm{e}_{11}}\}\ \exists E\,1(\{l_{\mathrm{t}_{12}}\}\ \mathsf{tell}(\mathrm{E} = [\mathrm{yes} \mid E\,1]))|| \\
\qquad \{l_{\mathrm{e}_{13}}\}\ \exists B\,1(\{l_{\mathrm{t}_{14}}\}\ \mathsf{tell}(B = [\mathrm{off} \mid B\,1])) \\
\quad \mathsf{else} \\
\qquad \{l_{\mathrm{e}_{15}}\}\ \exists E\,1(\{l_{\mathrm{t}_{16}}\}\ \mathsf{tell}(E = [\mathrm{no} \mid E\,1]))|| \\
\quad \{l_{\mathrm{p}_{17}}\}\ \mathsf{microwave_error}(D, B, E)))).
\end{array}
$$

Fig. 5. Example of a labeled tccp program: a simple error controller.

the execution of the program we are. The presence or absence of a label determines if the associated agent can be executed or not during the computation. The labeling process consists on the introduction of a different label for each occurrence of a language construct:

Definition 4

Let P be a specification, the labeled version P_l of P is defined as follows. The subindex $k \in \mathbb{N}$ corresponds to the number of labels introduced up to a given point. When the labeling process starts, $k = 0$ and each time that we introduce a new fresh label, k is incremented by one.

- If $P = \mathsf{stop}$ then $P_l = l_{\mathsf{stop}_k}\,\mathsf{stop}$.
- If $P = \mathsf{tell}(c)$ then $P_l = l_{\mathsf{tell}_k}\,\mathsf{tell}(c)$.
- If $P = \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$ then $P_l = l_{\mathsf{ask}_k} \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_l$.
- If $P = \mathsf{now}\,c\ \mathsf{then}\,A\ \mathsf{else}\,B$ then $P_l = l_{\mathsf{now}_k}\,\mathsf{now}\,c\ \mathsf{then}\,A_l\ \mathsf{else}\,B_l$.
- If $P = A||B$ then $P_l = l_{||_k}(A_l||B_l)$.
- If $P = \exists x\,A$ then $P_l = l_{e_k}\,\exists x\,A_l$.
- If $P = \mathsf{p}(x)$ then $P_l = l_{\mathrm{p}_k}\,\mathsf{p}(x)$.

The labeling of a declaration D of the form $\mathsf{p}(x)$:- A is defined as $l_{\mathrm{p}_i}\mathsf{p}(x)$:- A_l, called D_l. Finally, the labeled version of a program of the form $D.A$ is $D_l.A_l$.

In practice, we explore the tccp specification, and each time that we find an occurrence of a construct we introduce a new label which identifies such point of the program.

In Figure 5 we show the labeled version of the microwave error detection program showed in Figure 4. Note that the structure of the program has not changed, simply some labels have been added.

4.1.2 *The tccp structure*

The main point in the modeling phase is the construction of the graph structure which represents the system behavior. We define a new graph structure to represent the system. The tccp *Structure* is defined as a variant of the *Kripke Structure*. Intuitively, a Kripke Structure is a *finite* graph structure where there could be many initial nodes and each node is always related to another one (or to itself). Moreover, each state has associated a set of atomic propositions which are true in such state.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

The main difference between the two structures is that the definition of a state in the Kripke Structure follows the classical notion of state whereas in our structure, a state consists of a conjunction of constraints and intuitively it can be seen as a set of classical states.

Let us formally define our tccp Structure.

Definition 5
The set AP of atomic propositions is defined as the set of elements[2] of the cylindric constraint system $\mathscr{C}$ of the tccp language.

In the rest of the paper we abuse of notation by identifying the meaning of the terms *constraint*, *atomic proposition* and *element*. Next we define what a state of the tccp Structure is:

Definition 6 (*tccp State*)
Let AP be the atomic propositions in the tccp syntax and L be the set of all labels generated during the labeling process described above. We define the set of states as $S \subseteq 2^{AP} \times 2^{L}$.

Before the definition of the tccp Structure, we define the notion of equivalent states. For this, we need the classical notion of renaming of variables. Let $y_1, \ldots, y_n$ be n distinct variables. The substitution $\{x_1/y_1, \ldots, x_n/y_n\}$ is a *renaming*.

Definition 7 (*Equivalent States*)
Given two tccp states s and s', we say that the two states are equivalent if:

- the set of labels $l \subseteq L$ of s and the set of labels $l' \subseteq L$ coincide and,
- there exists a renaming γ of variables of the constraints in s which makes them syntactically identical to the set of constraints of s'

In Definition 8, we define the tccp Structure. Observe that the differences w.r.t. a Kripke Structure are the definition of state (in Definition 6) and the two labeling functions C and T which replace the labeling function L of the classical Kripke Structure.

Definition 8 (*tccp Structure*)
Let AP be a set of atomic propositions, we define a tccp Structure M over AP as a five tuple $M = (S, S_0, R, C, T)$ where

1. S is a finite set of states.
2. $S_0 \subseteq S$ is the set of initial states.
3. $R \subseteq S \times S$ is a transition relation.
4. $C : S \to 2^{AP}$ is the function that returns the set of atomic propositions in a given state.
5. $T : S \to 2^{L}$ as the function that returns the set of labels in a given state.

[2] See the definition in Section 2.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

We assume that a transition in the graph represents an increment of one time-unit in the system. Intuitively, C labels a state with the set of constraints true in such state. In other words, this function represents the new information that we know in a specific instant. T labels each state with the set of labels associated to agents that must be executed in the following time instant. In other words, T represents the point of execution in each instant (or state).

When two states s and s' are related by $R(s, s')$, it means that it is possible to reach the state s' from state s by executing the agents associated to the labels in $T(s)$ with the store $C(s)$ deriving as a result (by applying the renaming γ) the store $C(s')$ and the point of execution $T(s')$. In other words, given a state s and a renaming γ, we obtain a state s' whose store is $C(s') \cdot \gamma$.

Given a tccp Structure $Z = (S, S_0, R, C, T)$, we define $tr(Z)$ as the set of sequences of states of Z starting from an initial state and which are related by R:

$$tr(Z) = \{s \mid s = s_0 \cdot s_1 \cdots s_n \cdots \ \wedge s_0 \in S_0 \wedge \forall i \geqslant 0, \exists R(s_i, s_{i+1})\} \tag{1}$$

Which intuitively means that for each s_i, there exists a transition to the (renamed) state $s_{i+1} \cdot \gamma_i$.

4.1.3 Construction of the model

In this section we show how the tccp Structure that represents the system behavior is constructed from a labeled specification S in an automatic way. We present the pseudo-code of the necessary algorithms for the construction. Moreover, we show the complexity of such algorithms and explain the process from a theoretical point of view.

Intuitively, the construction evolves as follows. A process is composed by a set of clauses and a goal. A *specification* is a set of clauses. We describe how a specification (or declaration) can be transformed in a set of tccp Structures. Actually, for each different clause we construct a tccp Structure which is labeled with a unique name. This name can be used as one of the labels introduced in the program and is used when a procedure call refers to such clause. We consider that the declaration D_l of the form $\mathsf{p}(x)_l$:- A_l is a public information which is always available. We also assume that each label l_A is associated with the agent A.

The first algorithm that we show is the main procedure construct(D) (Figure 6) which, given a tccp declaration D_l of the form $l_p \mathsf{p}(x) : -A_l$, returns a tccp Structure $Q = \langle S, S_0, R, C, T \rangle$ representing the behavior of p.

We define globally a data type called *state* which represents a state of the tccp Structure. We assume that *store* is a conjunction of constraints and *label* is a set of labels in L.

```
state :
    st : store;
    ℓ[] : label;
```

In our pseudo-code, we use the *dot notation* to access to the components of a state. Moreover, we use the notation [] for lists of elements, thus $\ell[]$ is a list of labels. The $\aleph$ value is a possible value of a store denoting unsatisfiability.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

```
construct(input D : tccp declaration, output ⟨S, S0, R, C, T⟩ : tccp Structure)
    s : state;
    S', S0' : set of states;
    inf[] : store;
    lab[] : set of labels;
    j : int;

    S' = ∅;                              // ∅ denotes the empty set
    S0' = ∅;
    inf = instant(true, lA);             // lp p(x) : −Al
    lab = follows(lA);
    for j = 1 to sizeof(inf)
        if inf[j] <> ℵ then
            s = create_node(inf[j], lab[j]);
            C(s) = inf[j];
            L(s) = lab[j];
            S' = S' ∪ {s};
    S0' = S';
    construct_ag(S', S0', R', C', T');
    S = S'; S0 = S0'; R = R';
    C = C'; T = T';
```

Fig. 6. Description of the construction algorithm.

Finally, we simplify the treatment of functions C and T. Although we do not mention them in the algorithm itself, these functions correspond to the two components of the state structure of the algorithm. We also write $R(n, n')$ to describe that nodes n and n' are related.

Roughly speaking, in this algorithm the tccp Structure is initialized and the set of initial states is created. Then the function construct_ag (Figure 7) is called. This function iteratively completes the construction. Functions instant and follows are two auxiliary procedures used during the construction of the tccp Structure. We show them below.

Now we show (Figure 7) the construct_ag procedure, which uses two more auxiliary functions: the find(s, S) function, which returns a reference to the state in S which coincides (modulo renaming of variables) with s, and the perm function which, given two states, returns the necessary renamings which make them equivalent.

Given a label ll, follows(ll) returns the list which contains the labels associated to the agents that must be analyzed in the following time instant. Each element of the list corresponds to a different possible behavior of the system. For example, in the case of a conditional agent, the initial part of the list corresponds to the possible behaviors when the guard of the agent is satisfied, and the final part of the list corresponds to the case when it is not satisfied. Therefore, if two or more conditional agents are nested, then all the possible behaviors depending on the first *then* part will appear before those of the *else* part in the list. Since tccp restricts the number of nested conditional agents in a program, we can ensure that this algorithm terminates and the list of sets of labels is finite.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

```
construct_ag(input/output S[] : state; input S0[] : state
             input/output R : relation, C, T : function)
    stat1, stat2 : state;
    s[], acc[] : state
    inf[] : store;
    lab[] : set of labels;
    rn : renaming of variables;
    j,k : int;

    acc = S;
    j = 0;
    while acc <> ∅ do
        stat1 = select(acc);
        acc = remove(acc, stat1);
        inf = instant(stat1.st, stat1.ℓ);
        lab = follows(stat1.ℓ);
        for k=1 to sizeof(inf)
            if inf[k] <> ℵ then
                s[j] = create_node(inf[k], lab[k]);
                stat2 = find(s[j], S);
                if (stat2) then // there exists an equivalent state
                    rn = perm(s[j], stat2);
                    R(stat1, rn, stat2);
                else
                    R(stat1, {}, s[j]);
                    j = j + 1;
                    S = S ∪ {s[j]};
                    acc = acc ∪ {s[j]};
                    C[j] = inf[k];
                    L[j] = lab[k];
```

Fig. 7. Description of the construction algorithm for agents.

The follows algorithm uses two additional auxiliary functions, append and combine, which are functions that implement operations over lists: append(ℓ_1, ℓ_2) returns the concatenation of the two lists ℓ_1 and ℓ_2 whereas combine(ℓ_1, ℓ_2) constructs a new list whose elements consist of an element of ℓ_1 and an element of ℓ_2. For example, if $\ell_1 = \{\{l_1\}, \{l_2\}\}$ and $\ell_2 = \{\{l_3\}\}$, then the result of combine(ℓ_1, ℓ_2) is the list $\{\{l_1, l_3\}, \{l_2, l_3\}\}$.

We can show that the complexity of the algorithm showed in Figure 8 is exponential in the maximum number of nested agents in the specification. The high complexity is a theoretical case which does not occur in practice. We think that the complexity in practical cases should be semi-linear on average.

Lemma 1

The time complexity for the algorithm follows(A) presented in Figure 8 is $O(n * 2^m)$ where m is the maximum number of nested agents and n is the size of the list returned by follows(A).

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

```
list_of_sets_of_stores follows(ll : label)
    ℓ[], ℓ1[], ℓ2[] : set_of_labels;
    n, i, j : int;

    case A of                    // we assume that A is the agent associated with ll.
        stop : ℓ[1] = {};
        tell(c) : ℓ[1] = {};
        Σ_{i=1}^{n} ask(c_i) → A_i : for j = 1 to n
                                         ℓ[j] = l_{A_j};
                                     ℓ[n + 1] = {ll};
        now c then B1 else B2 : ℓ1 = follows(l_{B1});
                                ℓ2 = follows(l_{B2});
                                ℓ = append(ℓ1, ℓ2);
        B1||B2 : ℓ = combine(follows(l_{B1}), follows(l_{B2}));
        ∃x B1 : ℓ = follows(l_{B1});
        p(x) : ℓ = {l_p};        // where l_p represents the label
                                 // of the tccp Structure constructed for p
    end case;
    return ℓ;
```

Fig. 8. Description of the auxiliary algorithm follows(ll).

Proof

First of all, we know that the agent A has a finite number of nested agents. Moreover, we can see that the cost of the algorithm in the case of Tell and Stop agents is constant since follows(A) $= \{\}$ in such cases. The cost is constant also in the case of Procedure Call agents since follows(p(x)) returns a single label. For the Choice agent, the cost depends on the number of asks contained in the agent. Therefore, given the agent $\sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$, the cost will be $n+1$. In addition, we know that the maximum number of nested recursive calls is 2^m which corresponds to the worst case: when every nested agent is a parallel or conditional agent. Note that in these cases, the functions combine or append are used. These are indeed the expensive operations which we count. We assume that the cost of these functions is linear in the size of the resulting list.

Thus, the time complexity of the worst case is $O(n * 2^m)$. □

Next we show the second auxiliary function needed during the automatic construction of the model (see Figure 9). Given a store and a label, instant(c, ll) returns a list of stores which corresponds to the information which can be computed instantaneously (i.e., before the following time instant) by executing the agents associated with the label ll. In this algorithm we have marked the negation not(c) with a star to indicate that the semantics of negation is defined as the non satisfiability of c instead of the satisfiability of $\neg c$. The instant procedure uses the auxiliary function flat(st,ll) (Figure 10) which adds the constraint st to each element of the list ll returning a simple list of stores. If st is inconsistent with any element of the list, then the value of the element is set to $\aleph$.

It is easy to see that the time complexity of flat is linear on the size of the list.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

```
list_of_stores instant(input st : store, ll : label)
    s[], s1[], s2[]: store;
    j: int;

    case A of                          // we assume that ll is associated to the agent A
        abort : s[1] = true;
        tell(c) : s[1] = c;
        ∑_{i=1}^{n} ask(d_i) → A_i : for j = 1 to n
                                         s[j] = {d_i};
                                     s[n+1] = true;
        now d then B1 else B2 : s1 = flat(st, instant(st ⊔ d, l_{B1}));
                                s2 = flat(st, instant(not*(d) ⊔ st, l_{B2}));
                                s = append(s1, s2);
        B1||B2 : s = combine(instant(st, l_{B1}), instant(st, l_{B2}))
        ∃x B1 : s[1] = {st[y/x]} ⊔ instant(st, l_{B1})  //where y is a fresh variable
        p(x) : s[1] = true;            // where p(x) :: {l_{B1}}B1 is a
                                       // clause of the specification
    end case;
    return s;
```

Fig. 9. Description of the auxiliary algorithm instant(st, ll).

```
list_of_stores flat(input st : store, ll[] : store)
    s[]: store;
    j: int;

    for j = 1 to sizeof(ll)
        if ll[j] ⊔ st = false then
            s[j] = ℵ;
        else
            s[j] = ll[j] ⊔ st;
    return s;
```

Fig. 10. Description of the auxiliary algorithm flat(st,ll).

Lemma 2

The time complexity for the algorithm flat(c,ll) presented in Figure 10 is $O(n)$ where n is the number of elements in the list ll.

Proof

The proof is trivial since we iterate n times over the elements of the list. □

The complexity of the algorithm instant showed above is exponential in the maximum number of nested agents in the specification. Note that also in this case, this is a theoretical case which may only occur very rarely in practice. We think that the complexity in practical cases should be semi-linear on average.

Lemma 3

The time complexity for the algorithm instant(st,A) presented in Figure 9 is $O(n * 2^m + 2n)$ where m is the maximum number of nested agents and n is the cardinality of the list of stores returned by instant(st,A).

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

Proof
We know that the agent A has a finite number of nested agents. We also know that if the agent is a Stop, Tell or Procedure Call agent, then the cost of the function is constant. If A is a Choice agent, then we have a linear cost, in particular we have $O(n+1)$ since there is an iterative loop over the number n of guards in the Choice.

Now let us consider the three remaining cases. For both the Conditional and the Parallel agents we have two recursive calls, whereas for the Hiding agent we have a single recursive call. We assume that the combine and append functions are linear in the size of the two lists passed as argument (i.e., we take $O(n)$ where n is the number of elements in the resulting list).

Therefore, we can say that the upper-bound for the global complexity of the algorithm is $O(n * 2^m + 2n)$ where m is the maximum number of nested agents. □

Now we can analyze the complexity of the construct algorithm. First of all, we state the complexity for the construct_ag function.

Lemma 4
The time complexity for the algorithm construct_ag(S, S_0, s, R, C, T) presented in Figure 7 is $O(c * m * 2^m)$ where m is the maximum number of nested agents, and c is the number of states in the model.

Proof
By Lemma 3 and Lemma 1 we know the complexity of the auxiliary functions. Moreover, we know that select and remove take linear time and we assume that create_node has constant complexity. We know that the while loop will be executed c times, where c is the number of different states in the model.

We can see that each time the loop is executed, we have one procedure call to each auxiliary function. Moreover, we have a for loop which is executed at most $m+1$ times. Therefore, the cost of the for loop is $O(m)$ and the cost of the while loop is $2c * (m * 2^m)$. We ensure the finiteness of the number of states since we know that there is a finite number of combinations of labels and constraints (which appear in the specification) modulo renaming. □

Theorem 1
The time complexity for the algorithm construct(D) presented in Figure 6 is $O(c * (2m * 2^m))$ where m is the maximum number of nested agents and n is the cardinality of the resulting list.

Proof
We know the cost of the auxiliary algorithms. Following the structure of the algorithm, we can see that there is one call to the construct_ag function. In addition, we have a procedure call to the algorithm instant and follows. Then, we have to add the cost of such algorithms: $O(2n * 2^m + c * (2m * 2^m))$. We have also a for loop which is executed at most m times. Therefore, we obtain the global complexity given in this result. □

Let us now explain intuitively the idea of algorithm showed in Figure 7. Each time an agent is analyzed, some actions are executed. In the following description we show the intuitions behind the formal definitions:

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

Stop $S \equiv \mathsf{stop}$. When we find a stop agent, we add no information to the store, insert a self-loop over the new node and instantiate the set of labels to the empty set since the construction must be concluded.

Tell $S \equiv \mathsf{tell}(c)$. The new information c is introduced into the store and the label associated to S is removed from the labels to be executed.

Choice $S \equiv \sum_{i=1}^{n} \mathsf{ask}(c_i) \rightarrow A_i$. This agent leads to a set of corresponding branches in the graph. We introduce at most $m+1$ branches with $m \leqslant n$, one for each possible successful ask guard. Note that if a c_i condition is not consistent with the store $C(s)$ then the corresponding branch will not be generated. For each new node s_i', we define the transition $R(s, s_i' \cdot \gamma)$ where γ is the renaming obtained when new nodes are generated, and we define an extra arc $R(s, s_{m+1} \cdot \gamma)$ that corresponds to the case when the store does not entail any condition c_i but the execution of concurrent agents proceed (if there are no concurrent agents or there exist but they cannot proceed, then $s_{m+1} = s$ thus a self-loop is introduced). Moreover, we do not introduce any additional information into the store and the labels are updated.

Conditional $S \equiv \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B$. The construction process in this case follows the same idea as for the choice operator: we define two new nodes (s_1' and s_2') that correspond to the two possible behaviors. The first branch corresponds to the case when the store entails c. It is added to the store the information that the agent A can generate in a single time instant. Also the set of labels is updated. The second branch is defined in a similar way.

Parallel $S \equiv A||B$. When a parallel agent is analyzed, the new node generated depends on the execution of the agents A and B in the present time instant. This means that the new store is defined as the union of the information obtained from the execution of A and B (if it is possible to execute them). Also the set of labels depends on these two agents.

Hiding $S \equiv \exists x\, A$. The behavior of the hiding agent is modeled in the graph construction by the introduction of the necessary renaming of variables in the store.

Procedure Call $S \equiv \mathsf{p}(X_1, \ldots, X_n)$. When a procedure call is reached we finish the process by introducing in s' a reference to the initial node of the tccp Structure for p. If there are more concurrent agents that must be analyzed, then we continue by considering the tccp Structure already generated for such clause (with the necessary renaming of variables). We link the current node s with a simplified copy of this piece of structure. The simplification consists in eliminating the branches whose condition is inconsistent with the constraints derived by the other (parallel) agents. Thus, the new node s' depends on the execution of the other concurrent agents and the body of the clause for p.

If there are two (or more) procedure calls in parallel the process is similar and as many nodes as different possible behaviors are generated.

To illustrate the construction process, in Figure 11 we present the construction of the tccp Structure for the program in Figure 5. Remember that this program simply detects if the door is open when the microwave works and in that case turns the system off and emits an error signal.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

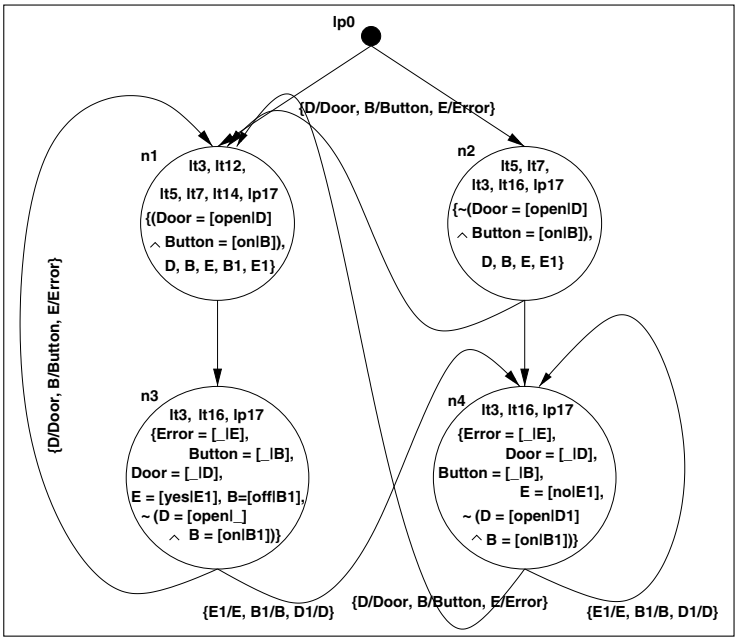


Fig. 11. Construction of the tccp Structures for the example shown in Figure 4.

We can see how, for the first time instant, two nodes corresponding to the two possible behaviors of the conditional agent have been generated in the specification (n1 and n2). Now look at the node n1 where we have that $L(\mathsf{n1}) = \{\mathsf{lt3}, \mathsf{lt12}, \mathsf{lt5}, \mathsf{lt7}, \mathsf{lt14}, \mathsf{lp17}\}$. This means that to continue with the graph construction we have to try to execute the agents associated with such labels. The tell agents update the store with the information that an error combination has been encountered and in the next time instant a *stop* signal will be present. This is important because when we try to execute the procedure call associated with lp_{17}, only one of the two possible branches can be followed.

When we generate new nodes and the corresponding connecting arcs we should consider formulas which are renamed apart. Note that if we find a node equal (up to renaming) to another one, a loop will be formed in the graph and the construction following this branch will terminate.

Next we show an additional example which may be useful to understand the construction. Given the program

$$\mathsf{p}(x) \text{:-} \exists y(\mathsf{tell}(x = f(y)) || p(y))$$

the constructed tccp Structure is shown in Figure 12. Note that, in each state, we store the new information added during a single time instant, thus the store of the program at a time instant k is given by the union of the information added along the path in the structure, after making k loops. For instance, after 3 time instants, the derived information is the following: $\{x = f(y), y = f(y')\}$. Roughly speaking, each time we loop on the second node, a renaming of variables which form the constraints in the store is performed. Thus, the renaming $\{x/y, y/y'\}$ where y' is

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

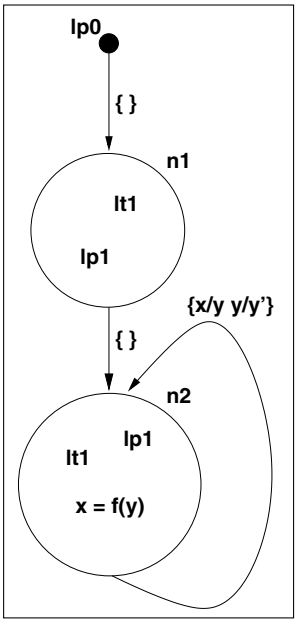


Fig. 12. Construction of the tccp structures.

a new variable, defines the new constraint $y = f(y')$. Following the syntax of the program, $x = f(y)$ and $y' = f(y'')$ are introduced by the tell agent in the first and second time instant respectively. Note that we show the store after 3 instants of time since the information produced by a tell agent in a given time instant (for example, the second), does not appear in the store till the following time instant. This is due to the fact that tell agents take one time instant.

4.1.4 Correctness and completeness

In this section we prove the correctness and completeness of the automatic construction of the model. We first introduce a function which extracts the information from the states of the tccp Structure. We define st as the set of sequences of the form $\{t \mid t = c_1 \cdot c_2 \cdots c_n \cdots\}$ where c_i is a finite constraint.

Definition 9
Given a tccp Structure Z and $s \in tr(Z)$ of the form $s_0 \cdot s_1 \cdot \ldots$, we define the function $\delta_s : tr(Z) \rightarrow st$ as follows:

$$\delta_s(s) = \begin{cases} C(s_0) & \text{if } s = s_0, \\ \delta(s_0) \cdot \delta_s(s') & \text{if } s = s_0 \cdot s', \end{cases}$$

where $\delta(s_i)$ is defined as

$$\delta(s_i) = \cup_{0 \leqslant j \leqslant i} C(s_j)$$

The extension of δ_s to sets of sequences is made in the obvious way.

The following theorem shows that the defined graph construction is correct and complete. In other terms it shows that the set of traces which correspond to the tccp

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

structure Z is the same given by the operational semantics of the tccp specification S.

Theorem 2
Let Z be the tccp Structure corresponding to the tccp specification S. Then the construction Z is correct and complete since

$$\delta_s(tr(Z)) = \mathcal{O}(S)$$

Proof
Let us first define an equivalence relation $\sim$ between configurations of the operational semantics presented in Figure 2 and graph states. Let $\sigma(\Gamma)$ be the store in configuration Γ, then we extend σ over sequences of configurations in the obvious way. In the graph, stores are 'extracted' by using function C. Then, we say that a configuration Γ corresponds to a tccp state s if $C(s) \vdash \sigma(\Gamma)$ and $\sigma(\Gamma) \vdash C(s)$, and the 'active agent' (namely one agent immediately reducible given the store in the current configuration) in Γ corresponds to that selected for reduction in s; we denote this by $\Gamma \sim s$.

A trace t of the form $s_0, \ldots, s_i, \ldots$ in a tccp Structure Z and a derivation (trace) $\gamma = \gamma_0, \ldots, \gamma_i, \ldots$ in the operational semantics of a specification S correspond iff $\delta_s(t) = \gamma$, i.e., $\forall i \ \delta(s_i) \sim \gamma_i$. We must prove that all (the partial) paths in the tccp Structure Z generated from the specification S have an equivalent trace in the operational semantics of S and vice-versa.

Let us first prove that $\delta_s(tr(Z)) \subseteq \mathcal{O}(S)$.

We proceed by induction on the length of the partial trace n in Z and on the structure of the agent A selected in step n. Note that each node in the tccp Structure has a finite number of successors, thus we can reason about all of them.

The basic case for $n = 0$ is trivial, since the tccp Structure Z is based on the same initial state s_0 considered in the operational semantics. Let us consider the inductive case, i.e., $n > 0$. Thus, let us consider the trace $s_0, \ldots, s_n$ in Z. We assume, by inductive hypothesis, that there exists a corresponding partial derivation $\gamma_0, \ldots, \gamma_n$ in $\mathcal{O}(S)$. We now prove that, if a further step is made in Z starting from s_n, it is possible to make a further step starting from γ_n in $\mathcal{O}(S)$ and the new states still correspond. Let $\pi = s_0, \ldots, s_n \in tr(Z)$ and let A be the active agent selected in s_n. We have to consider several cases corresponding to the possible structure of A.

Tell $A = \mathsf{tell}(c)$. Let $C(s_n) = d$ and $T(s_n) = \{l_{tell}\}$. Then, we have the trace $\gamma \in \mathcal{O}(A)$ with $\gamma = \gamma_0, \gamma_1, \ldots, \gamma_n$ and $\gamma_n = \langle A, d \rangle$, where s_n and γ_n correspond by inductive hypothesis. By the definition of the construction of the structure and the operational semantics we have that $C(s_{n+1}) = \{c \sqcup d\}$, $T(s_{n+1}) = \{\}$ and $\gamma_{n+1} = \langle \emptyset, c \sqcup d \rangle$ which correspond, thus $s_{n+1} \sim \gamma_{n+1}$.

Choice $A = \sum_{i=1}^{m} \mathsf{ask}(c_i) \rightarrow A_i$. Let $C(s_n) = d$ and $T(s_n) = \{l_{\mathrm{ask}}\}$. Then, we have the trace $\gamma \in \mathcal{O}(A)$ with $\gamma = \gamma_0, \gamma_1, \ldots, \gamma_n$ and $\gamma_n = \langle A, d \rangle$. By inductive hypothesis we have that $s_n \sim \gamma_n$. By definition of the construction of the structure and the operational semantics we have two cases: the first case is when there is no c_i such that $d \vdash c_i$, then in the construction of the tccp Structure there will be a loop, thus the state s_{n+1} actually is the state s_n whereas in the operational semantics

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

there is no possible transition. In this case we just take $\gamma_{n+1} = \gamma_n$, and clearly $s_{n+1} \sim \gamma_{n+1}$.

The second case is when there exists a c_i such that $d \vdash c_i$. This means that we have $C(s_{n+1}) = \{d\}$ and $T(s_{n+1}) = \{l_{A_i}\}$. It is clear that by selecting A in γ_n, we derive γ_{n+1}, which corresponds to s_{n+1}.

Conditional $A = \mathsf{now}\, c\, \mathsf{then}\, A_1\, \mathsf{else}\, A_2$. Let $C(s_n) = d$ and $T(s_n) = \{l_{\text{now}}\}$. Then, there exists a trace $\gamma \in \mathcal{O}(A)$ with $\gamma = \gamma_0, \gamma_1, \ldots, \gamma_n$ and $\gamma_n = \langle A, d\rangle$, such that, by inductive hypothesis, $s_n \sim \gamma_n$. By definition of the construction of the structure and the operational semantics we have two possible behaviors: either $d \vdash c$ or $d \nvdash c$. In the first case, $C(s_{n+1}) = \{d \sqcup \mathsf{instant}(d, l_{A_{n+1}})\}$and $T(s_{n+1}) = \mathsf{follows}(A_{n+1})$. On the other side, we have $\gamma_{n+1} = \langle A_1', d'\rangle$ where A_1' is the agent reached by the execution of A_1 and d' the new store with the information added by the execution of A_1. Clearly s_{n+1} and γ_{n+1} correspond. The case when $d \nvdash c$ is similar, considering A_2 for reduction.

Parallel $A = A_1 || A_2$. Let $C(s_n) = d$ and $T(s_n) = \{l_{||}\}$. Then, we have the trace $\gamma \in \mathcal{O}(A)$ with $\gamma = \gamma_0, \gamma_1, \ldots, \gamma_n$ and $\gamma_n = \langle A, d\rangle$. By inductive hypothesis $s_n \sim \gamma_n$. By definition of the construction of the structure and the operational semantics we have that $C(s_{n+1}) = \{d \sqcup \mathsf{instant}(d, l_{A_{n+1}}) \sqcup \mathsf{instant}(d, l_{A_2})\}$ and $T(s_1) = \{\mathsf{follows}(A_1) \cup \mathsf{follows}(A_2)\}$. Then, we have $\gamma_{n+1} = \langle A_1' || A_2', d'\rangle$ where A_1' (A_2') is the agent reached by the execution of A_1 (A_2) and d' is the new store with the information added by the execution of A_1 and A_2. Hence $s_{n+1} \sim \gamma_{n+1}$.

Exists $A = \exists x\, A_1$. Let $C(s_n) = d$ and $T(s_n) = \{l_e\}$. Then, we have the trace $\gamma \in \mathcal{O}(A)$ with $\gamma = \gamma_0, \gamma_1, \ldots, \gamma_n$ and $\gamma_n = \langle A, d\rangle$. By inductive hypothesis $s_n \sim \gamma_n$. We know that $C(s_{n+1}) = \{d \sqcup \mathsf{instant}(d, l_{A_1[y/x]})\}$. Note that $\mathsf{instant}(d, l_{A_1[y/x]})$ represents the information generated in one time step by the agent $A_1[y/x]$ which is the result of the application of the substitution y/x to the agent A_1 and $T(s_{n+1}) = \mathsf{follows}(A_1)$. y is a fresh variable, thus the information generated by A_1 involving such variable will not affect the rest of the system.

Now, following the operational semantics we derive that $\gamma_{n+1} = \langle \exists^{e'} xB, d \sqcup \exists_x e'\rangle$, where $\langle A_1, \exists_x d\rangle \rightarrow \langle B, e'\rangle$. Thus, we can identify e' with the information generated from agent A_1, and s_{n+1} and γ_{n+1} correspond.

Procedure Call $A = \mathsf{p}(X)$. Let $\mathsf{p}(X) : -B$ be a clause in the program (in the specification S). Let $C(s_n) = d$ and $T(s_n) = \{l_p\}$. By inductive hypothesis, there exists the trace $\gamma = \gamma_0, \gamma_1, \ldots, \gamma_n \in \mathcal{O}(A)$ and $\gamma_n = \langle A, d\rangle$. We have that $s_{n+1} = N$ where N is the first node of the tccp Structure constructed for $\mathsf{p}(X)$. We have that $C(s_{n+1}) = C(s_n)$ and $T(s_{n+1}) = l_B$. By expanding the procedure call in the operational semantics we get $\gamma_{n+1} = \langle B, d\rangle$, which clearly corresponds to s_{n+1}.

Now we have to prove that $\mathcal{O}(S) \subseteq \delta_s(tr(Z))$. This is completely analogous to the inclusion that we have proved. □

5 Specification of the property

In this section we present the logic which we use in our model checking algorithm. This is a temporal logic which has also the ability to handle constraints of a

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

given constraint system. In Boer *et al.* (2001), the authors presented a temporal logic for reasoning about tccp programs. In particular, it is an epistemic logic with two modalities, one representing the *knowledge* and the other one representing the *belief*. These two modalities allow us to reason with the input-output behavior of programs.

Given an atomic proposition c of the underlying constraint system, $\mathscr{K}(c)$ and $\mathscr{B}(c)$ are formulas of the logic which mean that c is *known* or c is *belief* respectively. In other words, $\mathscr{B}(c)$ holds if the process assumes that the environment provides c whereas $\mathscr{K}(c)$ holds if the information c is produced by the process itself.

The syntax of temporal formulas for this logic is shown below (see (Boer *et al.* 2001) for details):

Definition 10

Given an underlying constraint system with set of constraints $\mathscr{C}$, formulas of the temporal logic are defined by

$$\phi ::= \mathscr{K}(s) \mid \mathscr{B}(s) \mid \neg\phi \mid \phi \wedge \psi \mid \exists x\, \phi \mid \bigcirc \phi \mid \phi \,\mathscr{U}\, \psi$$

As for classical temporal logics, it is possible to define other logic operators such as the *always* or *eventually* operators from the basic ones. For example, if we want to express that a formula ϕ is satisfied at some point in the future, we write that $\Diamond\phi = true\,\mathscr{U}\,\phi$. To express that a formula ϕ is always satisfied, we can write that $\Box(\phi) = \neg(true\,\mathscr{U}\,\neg\phi)$. Moreover, as usual we denote by $\phi \rightarrow \psi$ the formula $\neg\phi \vee (\phi \wedge \psi)$.

A *reaction* is defined as a pair of constraints of the form $\langle c, d\rangle$ where c represents the input provided by the environment and d corresponds to the information produced by the process itself. Moreover, it holds that $d \geqslant c$ for every reaction, i.e., the output always contains the input.

The truth value of temporal formulas is defined with respect to *reactive sequences*. $\langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle\langle d, d\rangle$ denotes a reactive sequence which consists of a sequence of reactions. Each reaction in the sequence represents a computation step performed by an agent at time i. Intuitively each pair can be seen as the input-output behavior at time i.

Therefore, given a reactive sequence s we can define the truth values of formulas. The function $\mathsf{first}(s)$ returns the first reaction of a sequence, i.e., if $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle\langle d, d\rangle$ then $\mathsf{first}(s) = \langle c_1, d_1\rangle$. $\mathsf{next}(s)$ returns the sequence obtained by removing the first reaction of it, i.e., if $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle\langle d, d\rangle$ then $\mathsf{next}(s) = \langle c_2, d_2\rangle \cdots \langle c_n, d_n\rangle\langle d, d\rangle$.

We say that $\langle c, d\rangle \models \mathscr{B}(e)$ if $c \vdash e$, i.e., the reaction "believes" the constraint e if the first component of the reaction (c) entails e. Moreover, $\langle c, d\rangle \models \mathscr{K}(e)$ if $d \vdash e$, i.e., the reaction $\langle c, d\rangle$ "knows" the constraint e if its second component entails e.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

Table 1. *Logic operators properties*

$\mathscr{B}(c) \rightarrow \Box(\mathscr{B}(c))$
$\mathscr{K}(c) \rightarrow \Box(\mathscr{K}(c))$
$\mathscr{B}(c) \rightarrow \mathscr{K}(c)$
$\mathscr{K}(c) \rightarrow \bigcirc\mathscr{B}(c)$

Definition 11 (*by F. de Boer et al.*)
Let s be a timed reactive sequence and ϕ be a temporal formula. Then we define $s \models \phi$ by:

$s \models \mathscr{K}(c)$	if	$\mathsf{first}(s) \models \mathscr{K}(c)$
$s \models \mathscr{B}(c)$	if	$\mathsf{first}(s) \models \mathscr{B}(c)$
$s \models \neg\phi$	if	$s \not\models \phi$
$s \models \phi_1 \wedge \phi_2$	if	$s \models \phi_1$ and $s \models \phi_2$
$s \models \exists x \phi$	if	$s' \models \phi$ for some s' such that $\exists_x s = \exists_x s'$
$s \models \bigcirc\phi$	if	$\mathsf{next}(s) \models \phi$
$s \models \phi \mathscr{U} \psi$	if	for some $s' \leqslant s, s' \models \psi$ and for all $s' < s'' \leqslant s, s'' \models \phi$

where, for a sequence $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle$, we define the existential quantification $\exists_x s = \langle \exists_x c_1, \exists_x d_1\rangle \cdots \langle \exists_x c_n, \exists_x d_n\rangle$.

We say that a formula ϕ is valid ($\models \phi$) if and only if for every reactive sequence s, $s \models \phi$ holds. The reader can see that the modal operators $\mathscr{K}$ and $\mathscr{B}$ are monotonic w.r.t. the entailment relation of the underlying constraint system.

In this work we want to reason about tccp programs. Since the store of such programs evolves monotonically along the time, the notion of monotonically increasing reactive sequences is defined: let s be a reactive sequence of the form $\langle c_1, d_1\rangle \cdot \langle c_{n-1}, d_{n-1}\rangle\langle c_n, d_n\rangle$, then we say that s is monotonically increasing if it satisfies that $c_i \leqslant d_i$ and $d_j \leqslant c_{j+1}$ for each $i \in \{1, \ldots, n\}$ and $j \in \{1, \ldots, n-1\}$. From now on we consider only monotonically increasing reactive sequences. In Table 1 some properties of the logic operators are shown.

Therefore, whenever a constraint is believed in a specific time instant, then it will be believed also in all the following time instants. Moreover, if a given constraint is known at the present time instant, then it will be known at every time instant in the future.

Finally, we can define a relation between modal operators. In particular, we say that if a constraint c is believed at a specific time instant, then it is also known. Also, if the constraint c is known at a specific time instant, then it is believed at the following one.

The logic presented in this section can be seen as a kind of linear temporal logic. The reader can see that there are no quantifiers over alternative paths. It is considered that each instant of time has only one direct successor. If fact, if we compare this logic with the classical LTL logic (see (Clarke *et al.* 1999) for example) we can see that each temporal operator corresponds to a temporal operator from LTL.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

As we have said in the introduction, in model checking we assume a closed world in the sense that all the agents which can interact with the system are modeled. For this reason, the output in a time instant will always coincide with the input in the following time instant, i.e., it is not possible that other information different from the one generated by the model be introduced as an extra-input in any time instant. This mean that we can work with simple sequences of stores instead of working with sequences of reactions. We simply eliminate (or ignore) the second component of each reaction since it coincides with the fist one of the subsequent reaction.

From now, when we speak about sequences in the logic, we mean sequences of the form $s = s_0, s_1, \ldots$ where each s_i is a store and we omit the modal operator $\mathscr{K}$. The monotonic properties described above are maintained.

5.1 Some examples

Here we illustrate which kind of properties we are able to specify using this logic. We refer to the program example in Figure 4. Remember that such example models a very simplified program which controls the state of the door of a microwave.

We could check if it is true that when an error is detected, then the microwave has been turned-off. Actually, the error has occurred in the previous time instant since the door was open and the microwave was working, but the program can emit the error signal only in the following time instant, and at the same time the microwave should be turned-off.

The following formula represents such property.

$$\neg(\mathit{true}\ \mathscr{U} \neg\exists_{\{\mathrm{Error,E,Button,B}\}}(\mathrm{Error} = [\mathrm{no} \mid E] \vee (\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B]))) \tag{2}$$

It could seem that it is a complicate formula but if we think in terms of the always and eventually operators defined before, it becomes a very intuitive formula:

$$\Box\exists_{\{\mathrm{Error,E,Button,B}\}}(\mathrm{Error} = [\mathrm{no} \mid E] \vee (\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B]))$$

We can also model the property that the door will be eventually closed:

$$\Diamond\exists_{\{\mathrm{Door,D}\}}(\mathrm{Door} = [\mathrm{close} \mid D]) \tag{3}$$

Let us now remark the importance of the chosen logic in this work. We know that states of the tccp Structure represent only partial information. Therefore, if we want to check properties directly in the tccp Structure, then we need a logic able to handle partial information, as is the case of the logic presented in this section.

If we use any classical logic, we should consider each possible valuation of the variable values for each tccp state. In that case we had the same problem as in Falascxhi *et al.* (2000a, 2000b), i.e., we would not take advantage of the compact representation of the system that constraints can provide. Finally, the model-checking algorithm would not be effectively applicable for the state-explosion problem.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

6 The algorithm

The third and last phase of the model-checking technique is to define the algorithm which checks if a given temporal formula is satisfied by the model. The idea of the algorithm is similar to that for the classical tableau algorithm for the LTL model checking problem. The first thing is to construct the *closure* of the formula ϕ that we want to verify. Such closure is reminiscent of that of Fischer-Ladner's (Fischer and Ladner 1979).

Actually, if we intend to prove that the model satisfies the formula ϕ, then we construct the closure of the negated formula ($\neg\phi$). The atomic propositions of the logic are those of the underlying constraint system. The closure of the negated formula and the tccp Structure are used to construct a graph structure (called the *model-checking graph*). This graph structure consists of nodes of the form (q, Φ) where q is a state of the tccp Structure and Φ is a set of formulas from the closure of $\neg\phi$. The constructed graph structure allows us to verify if the property is satisfied or not by the system by using well known graph algorithms. In particular, we look for a path which starts from an initial state and reaches a *strongly connected component* (SCC) which satisfies some properties. If such path exists, then we can say that the property $\neg\phi$ is satisfied, thus ϕ is not satisfied in the model of the system. In this section we describe this process more in detail.

The construction of the graph combining the formula and the model might not terminate. It is for this reason that we use the interval of time which the user provides to the system. This interval imposes a time limit. If such time limit is reached, the system aborts the construction of the graph. The idea is that if this occurs, then we have obtained an over-approximation of the model, which nevertheless allows us to make useful verifications over the finite graph calculated.

6.1 The closure of the formula

The closure $CL(\phi)$ of a formula ϕ allows us to determine its truth value. Intuitively, it is the set of sub-formulas that can affect the truth value. This set is used classically to define tableaux algorithms where sub-formulas are evaluated as follows: simplest formulas are evaluated first, then more complex formulas are considered. Thus, we can say that the closure of ϕ ($CL(\phi)$) is the smallest set of formulas satisfying the following conditions:

- $\phi \in CL(\phi)$,
- $\neg\phi_1 \in CL(\phi)$ iff $\phi_1 \in CL(\phi)$,
- if $\phi_1 \wedge \phi_2 \in CL(\phi)$, then $\phi_1, \phi_2 \in CL(\phi)$,
- if $\exists x\, \phi_1 \in CL(\phi)$, then $\phi_1 \in CL(\phi)$,
- if $\bigcirc\phi_1 \in CL(\phi)$, then $\phi_1 \in CL(\phi)$,
- if $\neg\bigcirc\phi_1 \in CL(\phi)$, then $\bigcirc\neg\phi_1 \in CL(\phi)$,
- if $\phi_1 \mathcal{U} \phi_2 \in CL(\phi)$, then $\phi_1, \phi_2, \bigcirc\phi_1 \mathcal{U} \phi_2 \in CL(\phi)$.

Note that in the case of $\neg\bigcirc\,\phi_1$ it is necessary to introduce the formula $\bigcirc\,\neg\phi_1$ which cannot be generated by the other rules.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

Now we consider the microwave program example. The formula (2) for which we calculate the closure is that presented in the previous section.

Example 1

For the program showed in Figure 4 we construct the closure of the formula which we want to verify, starting from the negation of Formula (2). Note that we assume that $\neg\neg\phi = \phi$. We also change in the obvious way the disjunction operator into a conjunction:

$$true\,\mathcal{U}\,(\neg(\text{Error}=[\text{no} \mid E]) \wedge \neg(\text{Error}=[\text{yes} \mid E] \wedge \text{Button}=[\text{off} \mid B])) \tag{4}$$

Then, we show the closure of the formula. Note that the size of the set of formulas in the closure increases polynomially with the size of the formula (meaning the number of operators in the formula).

$$\begin{aligned}
CL(\chi) = \{&true\,\mathcal{U}\,(\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B])),\\
&true,\\
&false,\\
&\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B]),\\
&\neg(\text{Error} = [\text{no} \mid E]),\\
&\neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B]),\\
&\neg(\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B])),\\
&\text{Error} = [\text{no} \mid E],\\
&\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B],\\
&\text{Error} = [\text{yes} \mid E],\\
&\text{Button} = [\text{off} \mid B],\\
&\neg(\text{Error} = [\text{yes} \mid E]),\\
&\neg(\text{Button} = [\text{off} \mid B]),\\
&\bigcirc true\,\mathcal{U}\,(\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B])),\\
&\neg(\bigcirc true\,\mathcal{U}\,(\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B]))),\\
&\bigcirc\neg(true\,\mathcal{U}\,(\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B]))),\\
&\neg(true\,\mathcal{U}\,(\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B])))\\
&\}
\end{aligned}$$

6.2 *The model-checking graph*

Given a formula ϕ of the logic described in Section 5, and the tccp Structure Z constructed from the specification, the graph $G(\phi, Z)$ is defined as follows

Definition 12 (*Model-Checking Graph*)
Let ϕ be a formula, $CL(\phi)$ the closure of ϕ as defined in Section 6.1 and Z the tccp Structure constructed following the algorithm described in Section 4.1.3. A node n of the model-checking graph is formed by a pair of the form $(s_n, \mathcal{Q}_n)$ where s_n is a state of Z and $\mathcal{Q}_n$ is a subset of $CL(\phi)$ and the atomic propositions such that the following conditions are satisfied:

- for each atomic proposition p, $\mathcal{K}(p) \in \mathcal{Q}_n$ iff $p \in C(s_n)$,

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

- for every $\exists x\,\phi_1 \in CL(\phi), \exists x\,\phi_1 \in \mathcal{Q}_n$ iff $\exists_x \phi_1 \in C(s_n)$,
- for every $\phi_1 \in CL(\phi), \phi_1 \in \mathcal{Q}_n$ iff $\neg\phi_1 \notin \mathcal{Q}_n$,
- for every $\phi_1 \wedge \phi_2 \in CL(\phi), \phi_1 \wedge \phi_2 \in \mathcal{Q}_n$ iff $\phi_1 \in \mathcal{Q}_n$ and $\phi_2 \in \mathcal{Q}_n$,
- for every $\neg\circ\phi_1 \in CL(\phi), \neg\circ\phi_1 \in \mathcal{Q}_n$ iff $\circ\neg\phi_1 \in \mathcal{Q}_n$,
- for every $\phi_1\,\mathcal{U}\,\phi_2 \in CL(\phi), \phi_1\,\mathcal{U}\,\phi_2 \in \mathcal{Q}_n$ iff $\phi_2 \in \mathcal{Q}_n$ or $\phi_1, \circ\phi_1\,\mathcal{U}\,\phi_2 \in \mathcal{Q}_n$.

An edge in the graph is defined as follows: there will be an edge from one node $n_1 = (s_1, Q_1)$ to another node $n_2 = (s_2, Q_2)$ iff there is an arc from the node s_1 to the node s_2 in the tccp Structure and for every formula $\circ\phi_1 \in CL(\phi)$, $\circ\phi_1 \in Q_1$ iff $\phi_1 \in Q_2$.

Note that, in the definition above, when we take into consideration the set of arcs of the tccp Structure (when analyzing the formulas containing the next operator), we also consider the renaming that may label these arcs.

Intuitively, for each node of the model-checking graph, in $\mathcal{Q}$ we have the largest consistent set of formulas that is also consistent with the labelling function (the function C) of the tccp Structure. Moreover, two nodes of the graph are related if the temporal formulas in their $\mathcal{Q}$ sets are consistent.

For each node s_i of the tccp Structure many nodes are generated in the model-checking graph. All these nodes have as first component the state s_i and the second component consists of the different consistent sets of formulas derived from $C(s_i)$ and the closure of the formula.

Next we show an example to illustrate how the nodes of the model-checking graph are constructed. We construct the graph for the negation of the property since we intend to prove that there is no computation of the system which satisfies the negated property. This is equivalent to prove that the property is satisfied for all the computations.

Example 2

In this example we show some nodes of the graph which would result from our program example. We take the tccp Structure shown in Figure 11 and the closure set of the formula showed in the previous section.

Here we show two of the nodes generated for s_1 and one of the nodes generated for s_2.

$n_1 =$ (s_1, Q_1) where

$Q_1 =$ $\{$
$\mathrm{Door} = [\mathrm{open} \mid D] \wedge \mathrm{Button} = [\mathrm{on} \mid B]$,
$true,\ \mathrm{Error} = [\mathrm{no} \mid E]$,
$\neg(\mathrm{Button} = [\mathrm{off} \mid B])$,
$\neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B])$,
$\neg(\neg(\mathrm{Error} = [\mathrm{no} \mid E]) \wedge \neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B]))$,
$\circ true\ \mathcal{U}\ (\neg(\mathrm{Error} = [\mathrm{no} \mid E]) \wedge \neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B]))$,
$true\ \mathcal{U}\ (\neg(\mathrm{Error} = [\mathrm{no} \mid E]) \wedge \neg(\mathrm{Error} = [\mathrm{yes} \mid E] \wedge \mathrm{Button} = [\mathrm{off} \mid B]))$
$\}$

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

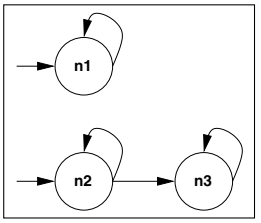


Fig. 13. A part of the model-checking graph for the tccp Structure shown in Figure 11 and the Formula (2).

$n_2 =$ (s_1, Q_2) where
$Q_2 =$ $\{$
$\text{Door} = [\text{open} \mid D] \wedge \text{Button} = [\text{on} \mid B]$,
true, $\text{Error} = [\text{yes} \mid E]$,
$\neg(\text{Button} = [\text{off} \mid B])$,
$\neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B])$,
$\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B])$,
$\mathit{true}\ \mathcal{U}\ (\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B]))$,
$\circ\mathit{true}\ \mathcal{U}\ (\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B]))$
$\}$
$n_3 =$ (s_2, Q_3) where
$Q_3 =$ $\{$
$\text{Error} = [\text{yes} \mid E]$, $\text{Button} = [\text{off} \mid B]$,
$\neg(\text{Door} = [\text{open} \mid D] \wedge \text{Button} = [\text{on} \mid B])$,
true,
$\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B]$,
$\neg(\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B]))$,
$\circ\mathit{true}\ \mathcal{U}\ (\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B]))$,
$\mathit{true}\ \mathcal{U}\ (\neg(\text{Error} = [\text{no} \mid E]) \wedge \neg(\text{Error} = [\text{yes} \mid E] \wedge \text{Button} = [\text{off} \mid B]))$
$\}$

Then, following the definition of the model-checking graph, we can define an arc from n_2 to n_3 since for each formula of the form $\circ\,\phi$ in the closure, if it is in Q_2 then ϕ is in Q_3. In figure 13, the reader can find a partial representation of the model-checking graph generated for the considered program and formula.

In this example, a brief time interval is sufficient to build the complete graph without approximation. During the construction, we can annotate how many steps are needed to reach each node from a root note, which determines the current instant of time. If such instant of time is equal to the time limit, then the construction is concluded and the graph obtained since that moment is given as output of the algorithm.

6.3 The searching algorithm

It is well known that in order to prove that a property is satisfied, it is possible to prove that there is no path satisfying the negation of the property. Thus, for verifying the formula ϕ, we construct the model-checking graph using the negation of ϕ and

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

the model of the system. Then we look for a sequence such that starting from the initial node of the graph, it reaches a *self-fulfilling strongly connected component* (SCC). Let us now give the formal definitions of SCC and self-fulfilling SCC.

Definition 13 (*Strongly Connected Component*)
Given a graph G, we define a Strongly Connected Component (SCC) C as a *maximal* subgraph of G such that every node in C is reachable from every other node in C along a directed path entirely contained within C.

We say that C is *nontrivial* iff either it has more than one node or it contains one node with a self-loop.

Then we can define a kind of strongly connected component. Actually, we will search for SCC satisfying the following properties in our model-checking algorithm.

Definition 14 (*Self-fulfilling SCC*)
Given a model-checking graph G, a self-fulfilling strongly connected component C is defined as a nontrivial strongly connected component in G which satisfies that for every node n in C and for every $\phi_1 \mathcal{U} \phi_2 \in Q_n$, there exists a node m in C such that $\phi_2 \in Q_m$, and vice-versa.

Now, let G be the model-checking graph generated following the steps described in Definition 12. We say that a sequence is an *eventually sequence* if it is an infinite path in G such that if there exists a node n in the path with $\phi_1 \mathcal{U} \phi_2 \in \mathcal{Q}_n$, then there exists another node n' in the same path reachable from n along the path, such that $\phi_2 \in \mathcal{Q}_{n'}$.

Moreover, we can prove the following result, which says that if we find a self-fulfilling strongly connected component in the corresponding model-checking graph, then the property represented by the formula is satisfied by the system. Our problem will be to prove that such self-fulfilling SCC does not exist[3].

Theorem 3
Let ϕ be a formula, Z a tccp Structure and $G(\phi, Z)$ the corresponding model-checking graph. If there exists a path in G, which satisfies a formula ϕ, from an initial node to a self-fulfilling strongly connected component, then the model Z satisfies the formula ϕ.

Proof
To prove this theorem we prove instead an equivalent result. We prove that if there exists an eventually sequence starting at an initial node $n = (s, \mathcal{Q}_n)$ such that the formula ϕ is in $\mathcal{Q}_n$, then the model satisfies the formula ϕ. This result is equivalent to the statement of the theorem since classical results (Clarke *et al.* 1999; Manna and Pnueli 1995) show that there exists an eventually sequence starting at a node $n = (s, \mathcal{Q}_n)$ if and only if there is a path in $G(\phi, Z)$ from n to a self-fulfilling SCC. We show that we can extend this result directly to our framework.

[3] Note that the result assumes that the construction of the graph has terminated before reaching the time limit provided by the user.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

Assume that we have an eventually sequence $n_1, n_2, \ldots$ where $n_1 = (s_1, \mathcal{Q}_{n_1})$, $n_2 = (s_2, \mathcal{Q}_{n_2})$, etc., starting with $n_1 = n$. This eventually sequence starts at node n_1 with $\phi \in \mathcal{Q}_{n_1}$. By definition, $\pi = s_1, s_2, \ldots$ is a path in the model Z starting at $s = s_1$. We want to show that $\pi \models \phi$. We will prove a stronger result: for every formula ψ in the closure of the formula ϕ ($\psi \in CL(\phi)$) and every $i \geqslant 0, \pi^i \models \psi$ iff $\psi \in \mathcal{Q}_{n_i}$. We follow the classical notations and by π^i with $i \geqslant 0$ we mean the suffix of the path π starting from the i-th component: $\pi^i = s_i, s_{i+1}, \ldots$. The proof proceeds by structural induction over the sub-formulas. There will be six cases corresponding to the six considered operators of the logic.

1. If ψ is an atomic formula, then by Definition 12 of node n_i, $\psi \in \mathcal{Q}_{n_i}$ iff $\psi \in C(s_i)$.
2. if $\psi = \exists x \chi$ then $\pi^i \models \psi$ iff $\psi \in C(s_i)$.
3. If $\psi = \neg\chi$ then $\pi^i \models \psi$ iff $\pi^i \not\models \chi$. By the inductive hypothesis, this holds iff $\chi \notin \mathcal{Q}_{n_i}$. By Definition 12, this guarantees that $\psi \in \mathcal{Q}_{n_i}$.
4. If $\psi = \chi_1 \wedge \chi_2$ then $\pi^i \models \psi$ iff $\pi^i \models \chi_1$ and $\pi^i \models \chi_2$. By the inductive hypothesis, this holds iff $\chi_1 \in \mathcal{Q}_{n_i}$ and $\chi_2 \in \mathcal{Q}_{n_i}$. By Definition 12 this is true iff $\psi \in \mathcal{Q}_{n_i}$.
5. if $\psi = \circ\chi$ then $\pi^i \models \psi$ iff $\pi^{i+1} \models \chi$. By the inductive hypothesis this holds iff $\chi \in \mathcal{Q}_{n_{i+1}}$. Since $((s_i, \mathcal{Q}_{n_i}), (s_{i+1}, \mathcal{Q}_{n_{i+1}})) \in R$, the above holds iff $\circ\chi \in \mathcal{Q}_{n_i}$.
6. if $\psi = \chi_1 \mathcal{W} \chi_2$ then by definition of an eventually sequence, there is some $j \geqslant i$ such that $\chi_2 \in \mathcal{Q}_{n_j}$. Since $\psi \in \mathcal{Q}_{n_i}$, the definition of a node implies that if $\chi_2 \notin \mathcal{Q}_{n_i}$, then $\chi_1 \in \mathcal{Q}_{n_i}$ and $\circ\psi \in \mathcal{Q}_{n_i}$. In this case, the definition of the transition relation of G implies that $\psi \in \mathcal{Q}_{n_{i+1}}$. It follows that for every $i \leqslant k < j$, $\chi_1 \in \mathcal{Q}_{n_k}$. By the inductive hypothesis, $\pi^j \models \chi_2$ and for every $i \leqslant k < j$, $\pi^k \models \chi_1$. Hence $\pi^i \models \psi$.
   Since $\pi^i \models \psi$, then there exists $j \geqslant i$ such that $\pi^j \models \chi_2$ and for all $i \leqslant k < j$, $\pi^k \models \chi_1$. We take the minimum j. By the inductive hypothesis, $\chi_2 \in \mathcal{Q}_{n_j}$ and for every $i \leqslant k < j$, $\chi_1 \in \mathcal{Q}_{n_k}$. Suppose $\psi \notin \mathcal{Q}_{n_i}$. Since $\chi_1 \in \mathcal{Q}_{n_i}$, by Definition 12 $\circ\psi \notin \mathcal{Q}_{n_i}$, which implies that $\circ\neg\psi \in \mathcal{Q}_{n_i}$. Now by definition of the transition relation of G, $\neg\psi \in \mathcal{Q}_{n_{i+1}}$, and hence $\psi \notin \mathcal{Q}_{n_{i+1}}$. Continuing the argument inductively, we would eventually find $\psi \notin \mathcal{Q}_{n_k}$, which is a contradiction since $\chi_2 \in \mathcal{Q}_{n_j}$.

This proves that if we have an eventually sequence, the model satisfies the formula ϕ. Now we have the classical result that can be applied to the graph G. If we look for an eventually sequence, we can instead look for a path from the initial node n to a self-fulfilling SCC. There are algorithms that implement this search with a complexity linear in the size of the graph and exponential in the size of the formula. □

For the complexity of the algorithm, we can see that the method is quite inefficient since it is based on the tableau algorithm for LTL. Note that such algorithm is PSPACE-complete. The important thing is the fact that we are dealing with a programming language and we can handle constraints as a powerful way to represent systems. Moreover, we obtain a similar complexity to the classical approach since we use a logic which is able to handle tccp states. If we had used a classical logic, the complexity would have increased too much since it would be necessary to unfold

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

the states of the graph structures to consider all the possible valuations of variables which could satisfy a given constraint.

7 Related work

We can find in the literature some related work which uses the notion of constraint to solve the automatic verification problem for infinite-state systems. For example, in Delzanno and Podelski (2001, 1999), the authors introduce a methodology to translate concurrent systems into CLP programs and verify safety and liveness properties over such CLP programs. Esparza and Melzer (1997) introduce a semi-decision algorithm that uses constraint programming to verify 1-safe Petri nets. In Delzanno and Podelski (2001, 1999), constraints are used as an abstract representation of sets of system states, and in Esparza and Melzer (1997) constraint programming is used for solving linear constraints in the implementation of the algorithm.

Constraints are useful for different purposes in software verification. They can be used in the checking algorithms as is done in Esparza and Melzer (1997); they can be used to model the problem as Delzanno and Podelsky do; and they can also be integrated into the specification language, that is used to model the system, as we do.

Regarding the systems that our approach is able to verify, we have seen that there are basically two main cases. The first case is when we are able to verify a system without the limitation on the time interval and the second case is when the time limit is reached. The first case corresponds to systems whose infinite nature comes from the fact that they use variables with an infinite domain. These systems are somehow similar to the ones that can be verified in Delzanno and Podelski (2001) for the properties of safety. In the second case we consider a large class of systems by using the time interval "approximation". If we reach the limit of time imposed by the user (obviously, if the user provides a too short time interval, then some systems of the first class end up in this second category) then we must stop the construction of the graph G at that point. Thus, we can verify the system, but we must consider that it is an approximation of the original system.

We note that there are some limitations in the tccp language since, for example, tccp is not able to model strong preemption while Delzanno and Podelski (2001) consider a language which can express this behavior.

In the last years many different extensions over time have been presented in the literature. There are approaches which extend the cc paradigm with a notion of *discrete* time (tccp, tcc (Saraswat *et al.* 1994) or ntcc (Valencia 2002)) and there is also an extension of the model with a notion of *continuous* notion of time (*hybrid* cc language (Gupta *et al.* 1998)). Regarding ntcc, in Valencia (2003), the author presented some decidability results with respect to such language. Those results show that it is possible to apply model checking to ntcc but no algorithm nor complexity studies are presented.

In Falaschi *et al.* (2000a, 2000b), a method to construct a structure was presented as a first step towards the definition of a model-checking technique for tcc. Nevertheless, the structure defined in Falaschi *et al.* (2000a, 2000b) to model tcc programs was

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

quite different from the structure defined in this work. Actually, in those works the modeling phase was defined in detail, giving only a brief description of the specification and the algorithmic phase.

The tcc structure had two kind of transitions: the *timed transitions* and the *normal transitions*. The set of states of the tcc Structure were defined in a way as similar to the tccp Structure and could also be seen as sets of classical states for a Kripke Structure. However, also in this case, classical model checking algorithms cannot be applied to tcc Structures. First of all because tcc Structures have two kind of transitions, and secondly because the algorithms cannot handle the notion of state of the graph structure. Note that in the tccp approach we have only one kind of transition relation, thus we have only one problem: how to handle states.

Another main difference between the tcc and the tccp Structure lies in the interpretation of branching points. Branching points in tcc Structures are due to the interleaving nature of the model. The *normal* transitions are instantaneous in the sense that they do not cause time steps. The branching points of the tccp Structure due to conditional agents can be viewed as the branching points which could appear in the quiescence points of the tcc Structure, i.e., when passing from one time instant to the following one. However, branching points of the tccp Structure due to Choice agents cannot be identified with anything in the tcc Structure since the tcc model is deterministic.

In Falaschi *et al.* (2000a, 2000b) the idea was to transform the tcc Structure into a Kripke structure, and hence the problem at this point was the huge number of states of the transformed structure. Essentially, we lost the possibility to take advantage of the compact representation that the notion of constraint provides.

In the tccp approach it is not necessary to eliminate the kind of transitions (since there is only one type). More important is the fact that it is not necessary to unfold the possible values of variables in order to define a model-checking method. Actually, we use a temporal logic which is able to handle the tccp states.

In Falaschi *et al.* (2001) a first approach to the problem of verification of hcc, which is similar to the problem for tccp was presented. The idea was the essentially similar, i.e., to define a structure able to represent the system behavior and to check properties over such structure. However, we just constructed the basic model which was transformed into a linear time automaton which could be given as input to a classical model checker such as HyTech.

8 Conclusions

In this work we have introduced a method that allows us to check properties from a temporal logic over reactive systems that are specified in the Temporal Concurrent Constraint Language defined in Boer *et al.* (2001). We have seen that we can adapt the classical method of LTL model checking to the logic presented in Boer *et al.* (2001) and the tccp Structure defined in this paper which models the system behavior. We have described a method that can handle generic programs written in tccp, which means that we are not restricting ourselves to finite-state systems. By using tccp we can define infinite-state systems that can be handled by the logic which

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

we have used. This epistemic logic allows us to work with constraints. Constraints can be seen as a compact representation of (possibly infinite) many states. In a previous work (Falaschi *et al.* 2000a,b), the authors have defined a structure which can help to verify a different class of reactive systems specified using another language from the ccp framework. Falaschi *et al.* (2000a, 2000b) defined a kind of structure that may seem similar to the tccp Structure but it is essentially different: the nodes and the arcs of the graph structure are interpreted in a different manner. Furthermore Falaschi *et al.* (2000a, 2000b) do not define any model-checking algorithm, rather they only concentrate on the modeling phase. We have proved that our verification method is correct and have illustrated how it works.

We plan to make a prototypical implementation of our system and test it on a set of benchmarks, such as protocol verification and verification of properties of concurrent systems like safety or liveness properties.

We also want to study how our method can be optimized in order to improve its efficiency. It is well known that this kind of classical model-checking algorithm is exponential in the size of the formula. Hence as future work we want to extend to our framework some efficient model-checking algorithms, such as symbolic model checking, for avoiding a complete construction of the graph.

References

Abdulla, P., Annichini, A., Bensalem, S., Bouajjani, A., Habermehl, P. and Lakhnech, Y. 1999. Verification of Infinite-State Systems by Combining Abstraction and Reachability Analysis. *Proceedings of the 11th International Conference on Computer Aided Verification*, N. Halbwachs and D. Peled, Eds. Lecture Notes in Computer Science, vol. 1633. Springer-Verlag, Berlin, pp. 146–159.

Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T. A., Ho, P.-H., Nicollin, X., Olivero, A., Sifakis, J. and Yovine, S. 1995. The algorithmic analysis of hybrid systems. *Theoretical Computer Science 138,* 1, 3–34.

Alur, R., Henzinger, T. and Wong-Toi, H. 1997. Symbolic analysis of hybrid systems. *Proceedings of the 37th IEEE Conference on Decision and Control*, pp. 702–707.

de Boer, F. S., Gabbrielli, M., and Meo, M. C. 2000. A Timed Concurrent Constraint Language. *Information and Computation 161*, 45–83.

de Boer, F. S., Gabbrielli, M. and Meo, M. C. 2001. A Temporal Logic for reasoning about Timed Concurrent Constraint Programs. *Proceedings of 8th International Symposium on Temporal Representation and Reasoning*, G. Smolka, Ed. IEEE Computer Society Press, pp. 227–233.

de Boer, F. S., Gabbrielli, M. and Meo, M. C. 2002. Proving Correctness of Timed Concurrent Constraint Programs. *ACM Transactions on Computational Logic (TOCL)*. To appear.

Boigelot, B. and Godefroid, P. 1996. Symbolic Verification of Communication Protocols with infinite State Spaces using QDDs. *Proceedings of the 8th International Conference on Computer Aided Verification*, R. Alur and T. A. Henzinger, Eds. Vol. 1102. Springer Verlag, Berlin, pp. 1–12.

Bouajjani, A., Esparza, J. and Maler, O. 1997. Reachability Analysis of Pushdown Automata: Application to Model-Checking. In *International Conference on Concurrency Theory*, A. Mazurkiewicz and J. Winkowski, Eds. Lecture Notes in Computer Science, vol. 1243. Springer-Verlag, Berlin, pp. 135–150.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

Bouajjani, A., Jonsson, B., Nilsson, M. and Touili, T. 2000. Regular Model Checking. In *Proceedings of the 12th International Conference on Computer Aided Verification*, E. A. Emerson and A. P. Sistla, Eds. Lecture Notes in Computer Science, vol. 1855. Springer-Verlag, 403–418.

Clarke, E. M. and Emerson, E. A. 1981. Design and synthesis of synchronization skeletons using branching time temporal logic. *Proceedings of Workshop on Logic of programs*. Lecture Notes in Computer Science, vol. 131. Springer-Verlag, Berlin, pp. 52–71.

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

Clarke, E. M., Grumberg, O., and Peled, D. 1999. *Model Checking*. The MIT Press, Cambridge, MA.

Cousot, P. and Halbwachs, N. 1978. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the 5th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM Press, pp. 84–97.

Dams, D. R. 1996. *Abstract Interpretation and Partition Refinement for Model Checking*. Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands. PhD thesis.

Delzanno, G. and Podelski, A. 1999. Model Checking in CLP. *Proceedings 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, R. Cleaveland, Ed. Lecture Notes in Computer Science, vol. 1579. Springer-Verlag, Berlin, pp. 223–239.

Delzanno, G. and Podelski, A. 2001. Constraint-based deductive model checking. *International Journal on Software Tools for Technology Transfer 3,* 3, 250–270.

Esparza, J. and Melzer, S. 1997. Model Checking LTL Using Constraint Programming. *Proceedings of the International Conference on Application and Theory of Petri Nets*, P. Azéma and G. Balbo, Eds. Lecture Notes in Computer Science, vol. 1248. Springer-Verlag, Berlin, pp. 1–20.

Falaschi, M., Policriti, A. and Villanueva, A. 2000a. Modeling Timed Concurrent systems in a Temporal Concurrent Constraint language. *Proceedings of the 2000 Joint Conference on Declarative Programming*. University of La Habana, La Habana, Cuba.

Falaschi, M., Policriti, A. and Villanueva, A. 2000b. Modeling Timed Concurrent systems in a Temporal Concurrent Constraint language - I. *Selected papers from 2000 Joint Conference on Declarative Programming*, A. Dovier, M. C. Meo, and A. Omicini, Eds. Electronic Notes in Theoretical Computer Science, vol. 48. Elsevier Science.

Falaschi, M., Policriti, A. and Villanueva, A. 2001. Time Limited Model Checking. *Proceedings of International Workshop on Specification Analysis and Validation for Emerging Technologies in Computational Logic (SAVE'01)*, G. Delzanno, S. Etalle, and M. Gabbrielli, Eds.

Fischer, M. J. and Ladner, R. E. 1979. Propositional Dynamic Logic of Regular Programs. *Journal of Computer and System Sciences 18,* 2, 194–211.

Gupta, V., Jagadeesan, R., and Saraswat, V. A. 1998. Computing with continuous change. *Science of Computer Programming 30,* 1–2, 3–49.

Hughes, G. E. and Creswell, M. J. 1968. *Introduction to Modal Logic*. Methuen and Co.

Kesten, Y., Maler, O., Marcus, M., Pnueli, A. and Shahar, E. 1997. Symbolic model checking with rich assertional languages. In *Proceedings of the 9th International Conference on Computer Aided Verification*, O. Grumberg, Ed. Lecture Notes in Computer Science, vol. 1254. Springer-Verlag, pp. 424–435.

Loiseaux, C., Graf, S., Sifakis, J., Bouajjani, A., and Bensalem, S. 1995. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal Methods in System Design 6,* 1, 11–44.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press

Manna, Z. and Pnueli, A. 1995. *Temporal Verification of Reactive Systems. Safety*. Springer-Verlag.

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

Nielsen, M., Palamidessi, C. and Valencia, F. 2002. Temporal Concurrent Constraint Programming: Denotation, Logic and Applications. *Nordic Journal of Computing 1*, 145–188.

Pnueli, A. and Shahar, E. 2000. Liveness and Acceleration in Parametrized Verification. *Proceedings of the 12th International Conference on Computer Aided Verification*, E. A. Emerson and A. P. Sistla, Eds. Lecture Notes in Computer Science, vol. 1855. Springer-Verlag, pp. 328–343.

Quielle, J. P. and Sifakis, J. 1982. Specification and verification of concurrent systems in CESAR. *Proceedings of the 5th International Symposium on Programming*. Lecture Notes in Computer Science, vol. 137. Springer-Verlag, Berlin, pp. 337–350.

Saraswat, V. A. 1989. Concurrent Constraint Programming Languages. *PhD Thesis*, Carnegie-Mellon University.

Saraswat, V. A., Jagadeesan, R., and Gupta, V. 1994. Foundations of Timed Concurrent Constraint Programming. *Proceedings of 9th Annual IEEE Symposium on Logic in Computer Science*. IEEE, New York, pp. 71–80.

Saraswat, V. A. and Rinard, M. 1990. Concurrent Constraint Programming. *Proceedings of 17th Annual ACM Symposium on Principles of Programming Languages.* ACM Press, pp. 232–245.

Saraswat, V. A., Rinard, M. and Panangaden, P. 1991. Semantic Foundations of Concurrent Constraint Programming. *Proceedings of 18th Annual ACM Symposium on Principles of Programming Languages.* ACM Press, New York, pp. 333–352.

Valencia, F. 2002. Temporal Concurrent Constraint Programming. PhD thesis, BRICS, University of Aarhus.

Valencia, F. 2003. Timed Concurrent Constraint Programming: Decidability Results and their Application to LTL. In *Proceedings of the Nineteenth International Conference on Logic Programming (ICLP'03)*. Lecture Notes in Computer Science, vol. 2916. Springer-Verlag, pp. 422–437.

Villanueva, A. 2003. Model checking for the concurrent constraint paradigm. PhD thesis, University of Udine in cotutela with Technical University of Valencia.

https://doi.org/10.1017/S1471068406002675 Published online by Cambridge University Press