Verifying Real-Time Properties of tccp Programs*

M. Alpuente
U. Politécnica de Valencia
Camino de Vera s/n
46022 Valencia
alpuente@dsic.upv.es

M.M. Gallardo , E. Pimentel
U. de Málaga
Campus de Teatinos s/n
29071 Málaga
{gallardo,ernesto}@lcc.uma.es

A. Villanueva
U. Politécnica de Valencia
Camino de Vera s/n
46022 Valencia
villanue@dsic.upv.es

Abstract

The size and complexity of software systems are continuously increasing, which makes them difficult and labor-intensive to develop, test and evolve. Since concurrent systems are particularly hard to verify by hand, achieving effective, automated verification tools for concurrent software has become an important topic of research. *Model checking* is a popular automated verification technology which allows us to determine the properties of a software system and enables more thorough and less costly testing. In this work, we improve the *model-checking* methodology previously developed for the *timed concurrent constraint programming* language tccp so that more sophisticated, quantitative properties can be verified by the model–checking tools. First, we refine the notion of global store and the entailment relation which are inherent to the concurrent constraint paradigm, by formulating a notion of *just entailed constraint*. Then, we define a real-time extension to the linear-time temporal logic that is used to specify the software properties by annotating discrete-time marks to formulae. Finally, we illustrate by means of one example the improved ability to check real–time properties, which cannot be currently checked within previous model–checking frameworks for tccp.

*This work has been partially supported by the EU (FEDER) and the Spanish MEC under grant TIN2004-7943-C04, the Generalitat Valenciana under grant GV03/25, and the ICT for EU-India Cross-Cultural Dissemination ALA/95/23/2003/077-054 project.

Keywords: Timed Concurrent Constraint Paradigm, Model Checking, Temporal Logic

1 Introduction

The ever-growing size and sophistication of software systems requires more powerful, automated analysis and verification tools that are able to improve the reliability of programs. When considering concurrent software, the problem of verification becomes even more acute, since correctness is more elusive to capture by any but very precise formal tools. Linear temporal logic, as it is used in model–checking procedures [7, 9], has been proven to be very appropriate for the verification of concurrent software. One of its attractive features is the qualitative representation of time, which is based more on the notion of precedence of events than on metric description.

The main problem that model–checking methodologies have to face is the traditional state-explosion problem which makes them inapplicable to large size systems. In the *concurrent constraint* paradigm ccp [10], the notion of store-as-valuation is substituted by the notion of store-as-constraint, so that very compact state representations are obtained. This makes tccp (the *timed concurrent constraint programming* language of [5] which extends ccp with a discrete notion of time and a suitable mechanism to model timeouts and preemptions) especially appropriate for analyzing timing properties of concurrent systems by model checking.

Two important features of tccp are the monotonicity of the store and the maximal parallelism of processes in execution. In other words, the store expands monotonically through time, and all parallel processes are run concurrently at each time instant. The change of state (the value of variables, in the imperative sense) can be modeled in tccp (as in other declarative languages) by using streams. One stream (implemented as a logical list) is associated to each single variable; then, each element of the list represents the value of the variable at a given time instant.

In order to specify the desired properties of the systems, both the tccp model–checking framework defined in [8] and the symbolic and abstract algorithms of [1] and [2, 3] rely on the linear temporal logic LTL of [6], which is especially tailored to reason with constraints. The main limitation of this framework is caused by the monotonicity inherent to tccp stores. Roughly speaking, information is incrementally recorded in a disorder bag (the store) so that the relative order in which two pieces of information are stored is unavoidably lost. Moreover, it is also difficult to recognize whether two variables correspond to the same incarnation of recursive procedure calls, unless some ad-hoc "packaging" predicates are explicitly introduced in the tccp code, as in [3]. As a consequence of these drawbacks of the previous framework, it is very hard to deal with quantitative temporal properties regarding the relative precedence among tccp events such as: "from the time instant in which $y = 2$ on, x will be always positive".

In this paper, we improve the existing model–checking technology for tccp so that sophisticated timing properties regarding temporal ordering can be naturally verified. First, we supply the global store with a suitable structure which allows us to recognize the information that is added at each time instant. A new notion of constraint entailment is also provided for structured stores so that the resulting computational model is proven equivalent to the original one. In order to improve the computational power of the model, we then introduce new temporal operators for the logic LTL that better exploit the structure of stores. Differently to the the *metric* temporal logics of [4] that annotates next-state and strong-until operators with nonnegative integer time points, we introduce discrete-time marks to formulae instead, which suffices to model synchronous real-time systems. This makes it also different from [4], which was devised to model dense time with discrete clocks.

The paper is organized as follows. In Section 2 we briefly introduce the essentials of model-checking methods for tccp, as defined in [1, 2, 3, 8]. In Section 4, we formalize the notion of structured store. In Section 5, we extend the LTL logic of [6] so that the ability to reason about temporal events is improved.

2 Model Checking for tccp

Model checking is an automatic technique for verifying finite state concurrent systems [9]. It consists of three main tasks:

1. Modeling the system to be analyzed using a *modeling language*. Since the model constructed represents a nondeterministic concurrent system, its execution will typically exhibit many different paths, which are usually formalized using a trace-based operational semantics for the modeling language. The constructed model should be an abstraction of the original system in order to diminish its complexity while preserving the process interactions that are critical for the proof of correctness.

2. Specifying the desirable properties that the model must fulfill. Temporal logics (TL) such as linear temporal logic (LTL) or branching time logic (CTL) are frequently used to express both safety and liveness properties of concurrent systems.

3. Running an automatic verification technique to check the correctness of the model w.r.t. a specific temporal property. Model checking algorithms work by exhaustively inspecting the state space associated to the model, searching for traces

that do not satisfy the desirable property. If no trace is found, then it is proven that the model satisfies the property; otherwise, the trace that is found is reported as a counter-example.

The main limitation of model checking is known as the *state explosion problem*, which occurs when the model to be verified generates too many states to be recorded by the model-checking tool. Using tccp as a modeling language partially mitigates this problem since tccp permits very compact state representations thanks to the use of constraints. On the other hand, the language introduces time aspects within the Concurrent Constraint paradigm, which makes it especially appropriate for analyzing timing properties of concurrent systems by model checking.

3 The modeling language tccp

In [5], the *Timed Concurrent Constraint* language (tccp for short) was defined as an extension of the Concurrent Constraint Programming language ccp [10]. In the cc paradigm, the notion of *store as valuation* is replaced by the notion of *store as constraint*. The computational model is based on a global store where constraints are accumulated, and on a set of agents that interact with the store. The model is parametric w.r.t. a cylindric constraint system $\mathcal{C}$ that is defined as follows.

Definition 1 *Let $\langle \mathcal{C}, \leq, \sqcup, true, false\rangle$ be a complete algebraic lattice where $\sqcup$ is the lub operation, and true, false are the least and the greatest elements of $\mathcal{C}$, respectively. Assume that Var is a denumerable set of variables, and for each $x \in Var$, there exists a function $\exists_x : \mathcal{C} \to \mathcal{C}$ such that, for each $u, v \in \mathcal{C}$:*

1. $u \vdash \exists_x u$
2. $u \vdash v$ *then* $\exists_x u \vdash \exists_x v$
3. $\exists_x(u \sqcup \exists_x v) = \exists_x u \sqcup \exists_x v$
4. $\exists_x(\exists_y u) = \exists_y(\exists_x u)$

Then $\langle \mathcal{C}, \leq, \sqcup, true, false, Var, \exists\rangle$ is a cylindric constraint system.

We will use the entailment relation $\vdash$ instead of its inverse relation $\leq$. Formally, given $u, v \in \mathcal{C}$, $u \leq v \iff v \vdash u$.

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

1. $true \vdash \delta_{xx}$
2. If $y \neq x, z$ then $\delta_{xz} = \exists_x(\delta_{xy} \sqcup \delta_{yz})$.
3. If $x \neq y$ then $\delta_{xy} \sqcup \exists_x(v \sqcup \delta_{xy}) \vdash v$.

Diagonal elements allow us to hide variables that represent local variables, as well as to implement parameter passing among predicates. Thus, quantifier $\exists_x$ and diagonal elements δ_{xy} allow us to properly deal with variables in constraint systems.

In tccp, a new conditional agent (now c then A else A) is introduced (w.r.t. ccp) which makes it possible to model behaviors where the absence of information can cause the execution of a specific action. Intuitively, the execution of a tccp program evolves by asking and telling information to the store. Let us briefly recall the syntax of the language:

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

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

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

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

4 Introducing Explicit Time in tccp

In this section, we define the new computational model of the language. First, we define the new notion of store and entailment relation, and then we adapt the original operational semantics of the language to the new formulation.

4.1 The Structured Store

The store used by tccp can be viewed as a blackboard where information is continuously written and never canceled. As we have shown above, the problem is that we add information without keeping track of the insertion order so that we cannot recover the time instant when a constraint has been added, which is primordial to analyzing temporal properties.

The following definition provides a structure to the notion of store by using time as a "state counter". Intuitively, a structured store consists of a timed sequence of stores. Each store represents only the information added at a given time instant by the different processes that are run concurrently. Thus, we can observe and analyze the evolution of the structured store through time.

Definition 2 *(Structured Store) We define a* structured store *as an infinite indexed sequence of stores, i.e., an element of the domain* $\textsc{Store} = \mathcal{C}^\omega = \mathcal{C} \times \cdots$. *We denote the* i^{th} *component of a structured store* st *as* st_i.

Now, to work with the new structure, we redefine the notion of entailment relation and the least upper bound (lub) of constraints. Intuitively, the information stored up to a given time instant t is the lub of all the stores st_i in the sequence $0 \leq i \leq t$.

Definition 3 *(Entailment relation) Given a constraint* $c \in \mathcal{C}$ *and a structured store* $st \in$ Store, *the new entailment relation* $\vdash_t$ *is defined as*

$$st \vdash_t c \Leftrightarrow (\sqcup_{0 \leq i \leq t} st_i) \vdash c$$

We also need to adapt the mechanism for updating the store, since we want to add the information into the right time instant.

Definition 4 *(Update) Given a structured store* st *and a constraint* $c \in \mathcal{C}$, *the addition of* c *to the store* st *at the time instant* t, $st \sqcup_t c$, *is the structured store* st', *where each component* st'_i *is defined as*

$$st'_i = \begin{cases} st_t \sqcup c & \text{if } i = t \\ st_i & \text{otherwise} \end{cases}$$

Intuitively, the updated structured store coincides with the old one in all the components except for component t, where constraint c is added. Moreover, we define the union of structured stores as $(st \sqcup st')_i = st_i \sqcup st'_i \ \forall i \geq 0$

4.2 Operational Semantics augmented with Time

Now we adapt the original operational semantics of the language to the new notions of Store and constraint entailment given in Definitions 2 and 3.

In Figure 1, we show the new transition relation $\longrightarrow \in (A \times \textsc{Store} \times \mathbb{N})^2$ where A is the set of tccp agents recalled in Section 3, and $\mathbb{N}$ is the domain of naturals. We have augmented the configurations handled by the semantics with a parameter that represents the current time instant. As it will become apparent later, the introduction of this parameter is possible because tccp agents are totally synchronized. The idea is that, at each time instant, we introduce the information generated by the agents into the right component of the structured store. In particular, when a tell agent adds a constraint to the store, we update the structured store by introducing that information into the component that corresponds to the subsequent time instant.[1]

Given a tccp program P, an agent A_0, and an initial store $st_0 = st_{0_0} \cdot true^\omega \in$ Store,[2] the *timed operational semantics* of P w.r.t. the initial configuration $\langle A_0, st_0 \rangle$, is

$$\mathcal{O}_T(P)[\langle A_0, st_0 \rangle] = \{st = st_{0_0} \cdot st_{1_1} \cdot \cdots \in \textsc{Store} | \langle A_i, st_i \rangle_i \longrightarrow \langle A_{i+1}, st_{i+1} \rangle_{i+1} \text{ for } i \geq 0\}$$

[1] Note that, in the original semantics, the information added by the tell agent is also available only in the subsequent time instant.

[2] We have called the initial store st_0, thus st_{0_0} represents the first component of such store.

Thus, for each structured store $st_i \in \text{STORE}$ incrementally built during the execution, the operational semantics only records its i^{th} component st_{i_i}, which corresponds to the constraints added at the i^{th} time instant. We assume that each trace in $\mathcal{O}_T(P)[\langle A_0, st_0\rangle]$ is infinite (the last configuration is repeated indefinitely if necessary).

Recall that the original tccp operational semantics, which we denote as $\mathcal{O}$, produces monotonic sequences of stores, that is, traces of the form $s = s_0 \cdot s_1 \cdots$ where each $s_i \in \mathcal{C}$ and $\forall i \geq 0. s_{i+1} \vdash s_i$. In contrast, the sequences of stores produced by the refined operational semantics given in Figure 1 are non monotonic. As commented above, the i^{th} component just contains the set of constraints added at the time instant i.

Let us explain how to transform structured stores into standard stores. Consider the structured store $st = st_0 \cdot st_1 \cdot \cdots$, then the monotonic sequence of stores $mon(st) = s_0 \cdot s_1 \cdot \cdots$, where $s_0 = st_0$ and $s_i = \sqcup_{j \leq i} st_j$. On the other hand, the sequence of monotonic stores produced by the standard trace semantics of tccp can be considered as a structured store in the obvious way.

The following theorem provides a soundness and completeness result of the new operational semantics with respect to the original one.

Theorem 1 *Consider a tccp agent A and an initial store $s_0 \in \mathcal{C}$. Let $st^0 = s_0 \cdot true^\omega$. Then*

1. *If $st \in \mathcal{O}(P)_T[\langle A, st^0\rangle]$ then $mon(st) \in \mathcal{O}(P)[\langle A, s_0\rangle]$.*
2. *If $s \in \mathcal{O}(P)[\langle A, s_0\rangle]$ then $\exists st \in \mathcal{O}_T(P)[\langle A, st^0\rangle]$ such that $\forall i \geq 0. s_i \vdash st_i$.*

Proof 1 *Claim 1 follows trivially from the definition of $mon(s)$ given above. Since the computation tree is finitely branching, Claim 2 is easily proven by structural induction on the tccp program together with induction on the length of the trace st.*

5 Introducing Explicit Time in the Logic LTL

The linear temporal logic defined in [6] uses modalities for distinguishing between the information *assumed* by agents prior to their execution (which is supposed to be produced by the environment), and the information produced by the execution of agents (the *known* information). However, when analyzing programs by model checking, it is usual to assume that models are completely specified, i. e., the environment is considered a part of the model to be analyzed. Therefore, in this paper we consider a simplified version of [6] where we get rid of modalities.

Given a constraint system $(\mathcal{C}, \vdash)$, the syntax of the temporal formulae of [6] is

$$\phi ::= c \mid \neg\phi \mid \phi \wedge \phi \mid \exists x \phi \mid \bigcirc \phi \mid \phi \; \mathcal{U} \; \phi$$

The rest of the standard propositional connectives and linear temporal operators are defined in terms of the above operators in the usual way: $\phi_1 \vee \phi_2 = \neg(\neg\phi_1 \wedge \neg\phi_2)$, $\phi \rightarrow \psi = \neg\phi \vee \psi$, $\Diamond\phi = true \; \mathcal{U} \; \phi$ and $\Box\phi = \neg\Diamond\neg\phi$.

Definition 5 *Consider the constraint system $(\mathcal{C}, \vdash)$, and a sequence of stores $s = s_0 \cdot s_1 \cdots$. The truth value of temporal formulae is defined as follows, where $s^i = s_i \cdot s_{i+1} \cdots$ is the suffix sequence of s starting at store s_i:*

(1) $s^i \models c$	*iff* $s_i \vdash c$
(2) $s^i \models \neg\phi$	*iff* $s^i \not\models \phi$
(3) $s^i \models \phi_1 \wedge \phi_2$	*iff* $s^i \models \phi_1$ *and* $s^i \models \phi_2$
(4) $s^i \models \exists x \phi$	*iff* $s' \models \phi$, *for some* s' *such that* $\exists_x s^i = \exists_x s'$
(5) $s^i \models \bigcirc\phi$	*iff* $s^{i+1} \models \phi$
(6) $s^i \models \phi_1 \mathcal{U} \phi_2$	*iff* $\exists k \geq i. s^k \models \phi_2$ *and* $\forall i \leq j < k$, $s^j \models \phi_1$

5.1 Refined LTL logic

In the following, we take advantage of structured stores to extend the expressiveness of LTL when modeling tccp programs.

Definition 6 *Consider the constraint system $(\mathcal{C}^\omega, \vdash_t)$ and a structured store st. Given $t \in \mathbb{N}$, we define the timed satisfaction relation $\models_t$ as follows:*

R1	$\langle \mathsf{tell}(c), st\rangle_t \longrightarrow \langle \mathsf{stop}, st \sqcup_{t+1} c\rangle_{t+1}$	
R2	$\langle \sum_{i=0}^{n} \mathsf{ask}(c_i) \rightarrow A_i, st\rangle_t \longrightarrow \langle A_j, st\rangle_{t+1}$	if $0 \leq j \leq n$ and $st \vdash_t c_j$
R3	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}$	if $st \vdash_t c$
R4	$\dfrac{\langle B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B', st'\rangle_{t+1}}$	if $st \nvdash_t c$
R5	$\dfrac{\langle A, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}}$	if $st \vdash_t c$
R6	$\dfrac{\langle A, st\rangle_t \not\longrightarrow}{\langle \mathsf{now}\, c\, \mathsf{then}\, A\, \mathsf{else}\, B, st\rangle_t \longrightarrow \langle B, st\rangle_{t+1}}$	if $st \nvdash_t c$
R7	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1} \text{ and } \langle B, st\rangle_t \longrightarrow \langle B', st''\rangle_{t+1}}{\langle A \| B, st\rangle_t \longrightarrow \langle A' \| B', st' \sqcup st''\rangle_{t+1}}$	
R8	$\dfrac{\langle A, st\rangle_t \longrightarrow \langle A', st'\rangle_{t+1} \text{ and } \langle B, st\rangle_t \not\longrightarrow}{\langle A \| B, st\rangle_t \longrightarrow \langle A' \| B, st'\rangle_{t+1}}$	
R9	$\dfrac{\langle A, st_1 \sqcup \exists x\, st_2\rangle_t \longrightarrow \langle A', st'\rangle_{t+1}}{\langle \exists^{st_1} x\, A, st_2\rangle_t \longrightarrow \langle \exists^{st'} x\, A', st_2 \sqcup \exists x\, st'\rangle_{t+1}}$	
R10	$\langle p(x), st\rangle_t \longrightarrow \langle A, st\rangle_{t+1}$	if $p(x) : -A \in D$

Figure 1: Refined operational semantics of the language

(1') $st \models_t c$ *iff* $st \vdash_t c$

(2') $st \models_t \neg\phi$ *iff* $st \not\models_t \phi$

(3') $st \models_t \phi_1 \wedge \phi_2$ *iff* $st \models_t \phi_1$ *and* $st \models_t \phi_2$

(4') $st \models_t \exists x\phi$ *iff* $st' \models_t \phi$, *for some* st' *such that* $\exists_x st = \exists_x st'$

(5') $st \models_t \bigcirc\phi$ *iff* $st \models_{t+1} \phi$

(6') $st \models_t \phi_1 \mathcal{U} \phi_2$ *iff* $\exists i \geq t.st \models_i \phi_2$ *and* $\forall t \leq j < i$, $st \models_j \phi_1$

Note that subindex t in $\models_t$ is variable; it represents the time instant where the temporal formula is evaluated. In the original logic, formulas are evaluated making a recursion on stores. This is possible since traces contain stores which grow monotonically. However, in the new logic, we need all the stores in the sequence in order to retrieve the computed information. For this reason, we evaluate temporal formulas making a recursion on time.

The following proposition proves that the new satisfaction relation is *equivalent* to the original one. However the relation $\models_t$ allows us to easily introduce time in temporal formulas, as we will show later.

Proposition 1 *Given a* tccp *sequence of stores* $s = s_0 \cdots$, *a structured store* st, $i \in \mathbb{N}$ *and a temporal formula* ϕ, *then*

1. $s^i \models \phi \Leftrightarrow s \models_i \phi$
2. $st \models_i \phi \Leftrightarrow mon(st)^i \models \phi$

The new satisfaction relation $\models_t$ will be used for two main tasks: 1) to ask the accumulated store at a given time instant as seen above; 2) to define more refined constraints of the structured store as described in Section 5.3.

Let us illustrate the new satisfaction relation by means of an intuitive example. In [2], LTL was used to describe properties by using the constraints of the underlying constraint system as the atomic propositions for the logic. Assume that Figure 2 shows a structured store that is produced by the execution of a program that follows the new operational semantics given in Figure 1. In the example, streams X and Z range over $\mathbb{N}$, and Y is a natural variable. Moreover, $0 < n < m$ is the range of indexes of the structured store.

Note that LTL reasons about computation paths in terms of sequences of stores, which

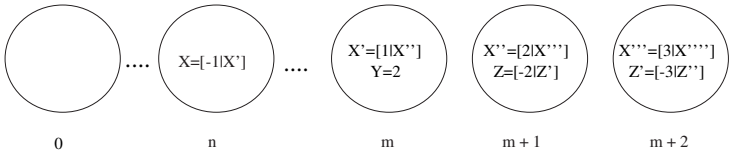

Figure 2: A Structured Store

is a notion that (almost) coincides with the notion of structured store. If we use the new entailment relation ($\vdash_t$) over structured stores, the two notions actually do coincide. In other words, the structured store, which is represented as an array, can be seen as a sequence of stores representing a specific execution.

Given the original non-timed tccp semantics $\mathcal{O}$ and the property P which establishes: "if $Y = 2$ then, from the next time instant on, the value of X will always be positive". P holds for the structured store shown in the example. However, we have problems expressing this property using temporal logic since X is a stream and we cannot explicitly use the notion of time. For instance, we could naively try to write P as the formula F given by

$$((Y = 2) \rightarrow \bigcirc\Box(\neg\exists X', N(X' = [N|\_] \wedge N \leq 0)))$$

However, this expression does not match the intended property for two main reasons:

- F does not hold since there exists an old value of X that is negative (specifically the one given by X in the example). This occurs because store evolves in a monotonic way; thus, the logic cannot distinguish if that instantiation occurred before or after the time instant m (when $Y = 2$).
- In addition, formula F is too restrictive since it imposes that the values of all streams (including those modeling variables different from variable X) must be positive from the time instant $m+1$. This also makes F false since variable Z' in time $m + 1$ has a negative value.

The new satisfaction relation $\models_t$, that takes time into account, allows us to improve how logic handles streams in such a way that the previous property P can be written simply as $Y = 2 \rightarrow \bigcirc\Box X > 0$.

In tccp, the notion of stream allows us to model imperative variables in the same way as logical lists are used in concurrent logic languages. We need streams to model the change of the value of variables through time because tccp traces contain stores which grow monotonically; otherwise, we would only model variables whose value remains unchanged. In the following section, we introduce a mechanism to drastically simplify the handling of streams. In particular, we show how to handle them as imperative variables, that is, the current value is directly accessible from the stream without recurring over the list.

5.2 Modeling Streams

Streams can be used to represent variables whose values may change during the program execution. Thus, at each time instant, the current value of a stream is the last value added to its tail as formalized in the next definition.

Definition 7 *(Current value of a variable) Let X be a stream, st a structured store and $t \in \mathbb{N}$. Then, A is* the value of X in st at instant t, *denoted by $st \models_t X = [\cdots A|X_s]$ (or simply $X = A$), iff $\exists m \geq 0$ such that:*

$$st \models_t X = [A_1, \cdots, A_m, A|X_s] \text{ and}$$
$$st \not\models_t \exists X'.X_s = [-|X']$$

We also need a measure for the length of the streams: given a structured store st, the length of stream X in st at time t is m, in symbols $length(X, st_t) = m$ iff $st \models_t X = [A_1, \cdots, A_{m-1}, A|X_s]$ and $st \models_t X = A$.

For instance, if we call 'st' the trace shown in Figure 2, then it holds that $st \models_m X = 1$ and $st \models_{m+1} Z = -2$.

The following notation is helpful.

Definition 8 *Assume that $cons(X_1, \cdots, X_n)$ is a constraint regarding the current values of streams $X_1, \cdots, X_n$, and st a structured store. Then, st satisfies $cons(X_1, \cdots, X_n)$ in the time instant t, in symbols $st \models_t cons(X_1, \cdots, X_n)$, iff $st \models_t X_1 = A_1 \wedge \cdots \wedge X_n = A_n \wedge cons(A_1, \cdots, A_n)$.*

For instance, considering the sequence of stores in Figure 2, it holds that:

$$st \models_{m+1} X + Z = 0$$
$$st \models_{m+1} \bigcirc(X + Z) = 0$$

Note that, in the previous temporal formulae, we use the original names for streams X and Z, that is, the variable identifiers when variables were created. The auxiliary names created during the execution are hidden, which clearly simplifies the representation of the temporal formulas.

Now, considering the trace in Figure 2 again, it holds that $st \models_m Y = 2 \rightarrow \bigcirc\Box X > 0$ and this formula represents the property P described above in a very concise and exact way.

5.3 Just Entailed Constraints

Unfortunately, with this new logic, we cannot yet distinguish whether a given property is satisfied for the first time at a given time instant; i.e., whether or not it is a consequence of the information added to the store at a previous time instant. This is a desirable feature to have since we sometimes want to detect whether a specific situation is true as a consequence of an agent that has just been executed. This may help us to prevent an invalid behavior of the system when this event is detected in advance. On the other hand, by recording the time instant when a constraint has been added or entailed, we will be able to verify real-time properties, as shown in the next section.

The problem we face here is that although we have structured the store in such a way that the sequence of stores is not monotonic, we are dealing with a notion of entailment ($\vdash_t$) that makes the whole monotonic again. This is because we accumulate all the information collected up to a given instant of time.

In order to get rid of this monotonicity, we define a more refined version of the simple constraints. These new constraints are, in some sense, more demanding and more difficult to fulfill than the usual constraints. Formally, given a constraint $c \in \mathcal{C}$, we introduce the new constraint $\overline{c}$, which represents that constraint c is now true for the first time. Note that when $t = 0$, all constraints are *just entailed constraints.*

(1") $st \models_t \overline{c}$ iff $st \models_t c$ and $st \not\models_{t-1} c$
(1"') $st \models_t \overline{cons(X_1, \cdots, X_n)}$ iff
$st \models_t cons(X_1, \cdots, X_n)$ and $\exists 1 \leq i \leq n.$
$length(X_i, st_t) > length(X_i, st_{t-1})$

6 Towards a Real-Time Logic

In this section, we extend the previously outlined temporal logic of [6] by annotating discrete time marks to formulae. To do this, we need some previous technical definitions.

Consider the *timed cylindric constraint system* $\langle \mathcal{C}_T, \leq_T, true, false, Var_T, \exists \rangle$ where $\mathcal{C}_T$ is a set of (unstructured) stores which introduces a distinguished class of timing constraints that consists of boolean expressions with the usual arithmetic operators. Var_T is an infinite set of variables used only to record times, such that $Var \cap Var_T = \emptyset$, Var being the set of variables used in tccp programs. Let us denote with $\vdash_T$ the corresponding entailment relation. Roughly speaking, $\mathcal{C}_T$ stores will be constructed along the evaluation of temporal formulas to record the precise time instants where certain constraints of interest are proven. They will typically include expressions of the form $t = m$, or $t \leq t' + m$ where $t, t' \in Var_T$ and $m \in \mathbb{N}$.

Definition 9 *(Annotated Temporal Formulas) Let $\mathcal{F}$ be the set of temporal formulas constructed with the elements of $\mathcal{C} \cup \overline{\mathcal{C}}$, the usual boolean connectives and the temporal operators. An annotated formula is an element of*

$\mathcal{E} = \mathcal{F} \times \mathcal{C}_T \times Var_T$, *where the first component is a classic temporal logic formula, the second one is a timing constraint, and the last one is used to record the time instant when the temporal formula is proven.*

The semantics of annotated temporal formulas is formalized as follows. Consider $\langle st, \tau \rangle$, where st is a structured store, $\tau \in \mathcal{C}_T$, and $\langle \phi, r, t \rangle \in \mathcal{E}$. Then,

$\langle st, \tau \rangle \models_m \langle \phi, r, t \rangle \iff$
$st \models_m \phi$ and $\tau \sqcup \{t = m\} \vdash_T r$

$\langle st, \tau \rangle \models_m \neg\langle \phi, r, t \rangle \iff \langle st, \tau \rangle \not\models_m \langle \phi, r, t \rangle$

$\langle st, \tau \rangle \models_m \langle \phi_1, r_1, t_1 \rangle \wedge \langle \phi_2, r_2, t_2 \rangle \iff$
$\langle st, \tau \rangle \models_m \langle \phi_1, r_1, t_1 \rangle$ and
$\langle st, \tau \rangle \models_m \langle \phi_2, r_2, t_2 \rangle$

$\langle st, \tau \rangle \models_m \exists x \langle \phi, r, t \rangle \iff$
$\langle st', \tau \rangle \models_m \langle \phi, r, t \rangle$
for some st' such that $\exists_x st = \exists_x st'$.

$\langle st, \tau \rangle \models_m \bigcirc\langle \phi, r, t \rangle \iff$
$\langle st, \tau \sqcup \{t = m\}) \rangle \models_{m+1} \langle \phi, r, t' \rangle$
where $t' \in Var_T$ is a fresh variable

$\langle st, \tau \rangle \models_m \langle \phi, r, t_m \rangle \mathcal{U} \langle \phi', r', t' \rangle \iff$
$\exists k \geq m. \forall m \leq j < k.$
$\langle st, \tau \sqcup_{i=m}^{j-1} (t_i = i) \rangle \models_j \langle \phi, r, t_m \rangle$
$\langle st, \tau \sqcup_{i=m}^{k-1} (t_i = i) \models_k \langle \phi', r', t' \rangle$

7 Case study: A railway crossing

We illustrate the real time logic presented above by specifying some temporal properties of the following tccp program that models the railway crossing example [11]. The system consists of three agents: **train**, **gate** and gate **controller**. Each agent behaves as follows:

`train` Sends message **near** to the **controller** when it is approaching the crossing. It also sends the message **out** when it has passed through the crossing.

`controller` When it receives the **near** message from the **train**, sends the message **down** to the **gate** and expects the confirmation. Similarly, when it receives the **out** message, sends the message **up** and expects the confirmation again.

`gate` When it receives the **down** message from the **controller** agent, it changes its state to **down** and responds properly. It behaves similarly when it receives the message **up**.

```
train(toC,T) :- ∃ toC',toC", T',T"(
     ask(true) → train(toC,T) +
     ask(true) →
        tell(toC = [near|toC'])||
        ask (true)^300 →
            tell(T = [enter|T']) ||
            ask (true)^20 →
                tell(T' = [leave|T"]) ||
                tell(toC' = [out|toC"]) || train(toC",T"))

controller(toC,toG,fromG) :- ∃ toC', toG',fromG'(
     ask(toC=[near|-]) →
        tell(toC=[near|toC']) || tell(toG=[down|toG']) ||
        ask(fromG=[confirm|-]) →
           tell(fromG=[confirm|fromG']) ||
           controller(toC',toG',fromG')
     +
     ask(toC=[out|-]) →
        tell(toC=[out|toC']) || tell(toG=[up|toG']) ||
        ask(fromG=[confirm|-]) →
           tell(fromG=[confirm|fromG']) ||
           controller(toC',toG',fromG'))

gate(fromG,toG,G):- ∃ fromG',toG',G'(
     ask(toG = [down|-]) →
        tell(toG=[down|toG']) ||
        ask (true)^100 →
            tell(G = [down|G']) ||
            tell(fromG=[confirm|fromG']) ||
            gate(fromG',toG',G')
     +
     ask(toG = [up|-]) →
        tell(toG=[up|toG']) ||
        ask (true)^100 →
            tell(G = [up|G']) ||
            tell(fromG=[confirm|fromG']) ||
            gate(fromG',toG',G'))

init:- ∃ toC,T,ToG,fromG,G (train(toC,T)||
        controller(toC,toG,fromG)||
        gate(fromG,toG,G))
```

Figure 3: tccp model for a railway crossing

This problem can be modeled in tccp as shown in Figure 3. The timing information encoded in the example is the following: a) the **train** takes at least 300 seconds from triggering the **near** message to reach the crossing; b) the **train** takes at least 20 seconds to get across the crossing; and, c) the **gate** takes 100 seconds to get the gate into position following an instruction.

Note that we have implemented this timing information using **ask** sentences. For instance, $\texttt{ask(true)}^{100}$ represents a delay of 100 time units for the **gate** agent.

We can now specify different timed properties of the model by using the real-time logic

outlined in Section 6. For instance:

Property 1: "When the train is near the crossing, it takes less than 300 seconds to have the gate down".

$$\Box\langle\overline{toC = \texttt{near}}, true, t\rangle \rightarrow \bigcirc\Diamond\langle\overline{G = \texttt{down}}, t' \leq t + 300, t'\rangle$$

Property 2: "When the train enters the crossing, the gate is down, and it remains down at least 20 seconds"

$$\Box(\langle\overline{T = \texttt{enter}}, true, t_0\rangle \rightarrow \langle G = \texttt{down}, true, t_1\rangle\mathcal{U}\langle\overline{G = \texttt{up}}, t_0 + 20 \leq t_2, t_2\rangle)$$

The way which these properties are verified on tccp programs involves the construction of appropriate model checking algorithms taking into account how every layer of the structured store represents a clock tick. It is mostly technical and not difficult to adapt the model-checking methodology of [8] to deal with the new tccp model and real-time formulas. Due to lack of space, we do not include here the details.

8 Conclusions

In this work, we have shown how to introduce explicit time in the *timed concurrent constraint programming* language tccp. This has allowed us to improve the *model-checking* methodology previously developed for tccp so that more sophisticated, quantitative properties can be verified by model–checking tools. This was not immediate in the original framework mainly because tccp stores are unstructured. Hence, first we have structured the store and consistently adapted the operational semantics of the language as well as the logic LTL to work with time instants. Then we have presented a method to handle streams within LTL formulae in a simpler way, and introduced the notion of *just entailed* constraint in order to know whether a constraint, which didn't hold in the previous time instant, now holds. Finally, we have refined the LTL logic in order to model real-time properties, and we have illustrated it by means of an example.

As future work, we plan to explore the possibility to deal also with past real-time formulas.

References

[1] M. Alpuente, M. Falaschi, and A. Villanueva. A Symbolic Model checker for tccp Programs. In *Proc. of the Int. Works. on Rapid Integration of Software Ingeneering techniques (RISE'04)*, LNCS 3475, pages 45–56. Springer Verlag, 2005.

[2] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. Abstract Model Checking of tccp programs. In *Proc. of the 2nd Works. on Quantitative Aspects of Programming Languages (QAPL 2004)*, volume 112 of *ENTCS*, pages 19–36. Elsevier Science, 2004.

[3] M. Alpuente, M.M. Gallardo, E. Pimentel, and A. Villanueva. A Semantic Framework for the Abstract Model Checking of tccp programs. *Theoretical Computer Science*, to appear, 2005.

[4] R. Alur and T. A. Henzinger. A really temporal logic. *Journal of the ACM*, 41(1):181–204, 1994.

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

[6] F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Temporal Logic for reasoning about Timed Concurrent Constraint Programs. In *Proc. of 8th Int. Symposium on Temporal Representation and Reasoning*, pages 227–233. IEEE Computer Society Press, 2001.

[7] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. In *Proc. of the 10th Annual ACM Symposium on Principles of Programming Languages*, pages 117–126. ACM Press, 1983.

[8] M. Falaschi and A. Villanueva. Automatic Verification of Timed Concurrent Constraint programs. *Theory and Practice of Logic Programming*, to appear, 2005.

[9] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems - Specification.* Springer-Verlag, New York, 1992.

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

[11] S. Schneider. *Concurrent and Real-time Systems. The CSP Approach.* Wiley, 2000.