A Temporal Logic for reasoning about Timed Concurrent Constraint Programs

F.S. de Boer
Universiteit Utrecht
frankb@cs.uu.nl

M. Gabbrielli
Università di Udine
gabbri@dimi.uniud.it

M.C. Meo
Università di L'Aquila
meo@univaq.it

Abstract

A temporal logic is presented for reasoning about the correctness of timed concurrent constraint programs. The logic is based on epistemic modalities which express either what a process knows at a certain time or what a process believes about the results of the other processes. In terms of these epistemic modalities of "knowledge" and "belief" a compositional axiomatization is given of the reactive behaviour of timed concurrent constraint programs.

1 Introduction

Many computer applications, usually called reactive systems, involve time-critical aspects and often require a programmer to specify timing constraints such as, for example, that an input is required within a bounded period of time. Reactive systems include real-time systems which are subject to hard timing constraints (e.g. process controllers and signal processing systems). Many different formalisms have been developed to specify, verify and program reactive systems, including (several) timed process algebras (see for example [1, 10]), temporal logic (and its executable versions) [12, 9] and the concurrent synchronous languages ESTEREL, LUSTRE, SIGNAL and Statecharts which have been already been used in many industrial applications.

Inspired by these formalisms a different approach to specify and program reactive systems has recently emerged in the context of *concurrent constraint programming (ccp)* [14]. This is an asynchronous concurrent programming paradigm in which the idea of generating and satisfying constraints is central to the computing process. In [16, 4] timed extensions of the pure ccp formalism were studied. The resulting languages are built around the hypothesis of *bounded asynchrony* [16]: Computation takes a bounded period of time rather than being instantaneous (as it is in ESTEREL [2]) and the whole system evolves in cycles corresponding to time units. The languages defined in [15, 16] are deterministic ones, useful mainly for programming small "kernels" of real-time systems (in their standard version they can be compiled to finite state automata). However, non-determinism arises when considering large reactive systems involving several processes running on different processors and communicating via asynchronous links. These (timed) systems can be naturally specified and programmed by using a non-deterministic language, indeed all the existing timed process algebras and almost all the variants of Statecharts admit non-determinism. Therefore in [4] we investigated a non-deterministic timed ccp language, called *tccp*, that provides a (Turing powerful) natural extension of the ccp framework.

In this paper we introduce a temporal logic for reasoning about *timed reactive sequences*. A timed reactive sequence describes at each moment in time the reaction of a tccp process to the input of the external environment. A reaction is formalized as a pair of constraints $\langle c, d\rangle$, where c is the input given by the environment and d is the constraint produced by the process itself. The basic assertions of the temporal logic describe these reactions in terms of *epistemic modalities* which express either what a process *believes* or *assumes* about the inputs of the environment and what a process *knows* or *commits* to, i.e., has itself produced at one time-instant. These epistemic modalities of "belief" and "knowledge" provide then a kind assumption/commitment style of specification. The main result of this paper is a compositional proof system for reasoning about the correctness of *tccp* programs as specified by formulas in this temporal logic.

2 The tccp language

In this section we introduce the *tccp* language [4] and we define its operational semantics of by using a transition system. Since the starting point is ccp, we introduce first some basic notions related to this programming paradigm. The ccp languages are defined parametrically wrt to a given *constraint system*. The notion of constraint systems has been formalized in [14] according to the Scott's treatment of information systems and by using some notions from cylindric algebras in order to treat the hiding operator of the language (by using an abstract $\exists_x$ operator) and to model

parameter passing (by using diagonal elements). Here we only consider the resulting structure and (we refer to [14] for further details).

Definition 2.1 Let Var be a given (denumerable) set of variables with typical elements $x, y, z, \ldots$. A (cylindric) constraint system is a structure $\langle \mathcal{C}, \leq, \sqcup, true, false, Var, \exists_x, d_{xy} \rangle$ where $\langle \mathcal{C}, \leq, \sqcup, true, false \rangle$ is a complete algebraic lattice ($\sqcup$ is the lub operation, $true$, $false$ are the least and the greatest elements), for each $x \in Var$ there exists a function $\exists_x : \mathcal{C} \to \mathcal{C}$ and for each $x, y \in Var$ the set$\mathcal{C}$ contains the constraints d_{xy} (so called diagonal elements) such that the following axioms are satisfied

(i) $\exists_x(c) \leq c$
(ii) if $d \leq c$ then $\exists_x(d) \leq \exists_x(c)$,
(iii) $\exists_x(c \sqcup \exists_x(d)) = \exists_x(c) \sqcup \exists_x(d)$
(iv) $\exists_x(\exists_y(c)) = \exists_y(\exists_x(c))$
(v) $d_{xx} \leq true$
(vi) if $z \neq x, y$ then $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$,
(vii) if $x \neq y$ then $c \leq d_{xy} \sqcup \exists_x(c \sqcup d_{xy})$.

Note that if $\mathbf{C}$ models the equality theory, then the elements d_{xy} can be thought of as the formulas $x = y$. In the sequel we will identify a system $\mathbf{C}$ with its underlying set of constraints $\mathcal{C}$ and we will denote $\exists_x(c)$ by $\exists_x c$ with the convention that, in case of ambiguity, the scope of $\exists_x$ is limited to the first constraint sub-expression.

The basic idea underlying ccp is that computation progresses via monotonic accumulation of information in a global store. Information is produced by the concurrent and asynchronous activity of several agents which can add (*tell*) a constraint to the store. Dually, agents can also check (*ask*) whether a constraint is entailed by the store, thus allowing synchronization among different agents. Parallel composition in ccp is modeled by the interleaving of the basic actions of its components.

When querying the store for some information which is not present (yet) a ccp agent will simply suspend until the required information has arrived. In timed applications however often one cannot wait indefinitely for an event: In case a given time bound is exceeded (i.e. a *time-out* occurs), the wait should be interrupted and an alternative action should be taken. Moreover in some cases it is also necessary to abort an active process A and to start a process B when a specific event occurs (this is usually called *preemption* of A). In order to be able to specify these timed behaviours in ccp, we introduce a discrete global clock and assume that ask and $tell$ actions take one time-unit. Computation evolves in steps of one time-unit, so called clock-cycles. We consider action prefixing as the syntactic marker which distinguishes a time instant from the next one. Furthermore we make the assumption that parallel processes are executed on different processors, which implies that at each moment every enabled agent of the system is activated. This assumption gives rise to what is called *maximal parallelism*. The time in between two successive moments of the global clock intuitively corresponds to the response time of the underlying constraint system. Furthermore, on the basis of the above assumptions we introduce a timing construct of the form **now** c **then** A **else** B which can be interpreted as follows: If the constraint c is entailed by the store at the current time t then the above agent behaves as A at time t, otherwise it behaves as B at time t. As shown in [4, 15] this basic construct allows one to derive such timing mechanisms as time-out and preemption. Thus we end up with the following syntax of timed concurrent constraint programming.

Definition 2.2 [*tccp* Language [4]] Assuming a given cylindric constraint system $\mathbf{C}$ the syntax of *agents* is given by the following grammar:

$$\begin{array}{rl} A ::= & \mathbf{tell}(c) \mid \sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i \\ & \mathbf{now}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B \mid A \parallel B \mid \exists x A \mid p(x) \end{array}$$

where the c, c_i are supposed to be *finite constraints* (i.e. algebraic elements) in $\mathcal{C}$. A *tccp process* P is then an object of the form $D.A$, where D is a set of procedure declarations of the form $p(x) :: A$ and A is an agent.

Action prefixing is denoted by $\rightarrow$, non-determinism is introduced via the guarded choice construct $\sum_{i=1}^{n}$, parallel composition is denoted by $\parallel$, and a notion of locality is introduced by the agent $\exists x A$ which behaves like A with x considered local to A, thus hiding the information on x provided by the external environment. In order to simplify the notation, in the following we will omit the $\sum_{i=1}^{n}$ whenever $n = 1$.

The operational model of *tccp* can be formally described by a transition system $T = (Conf, \longrightarrow)$ where we assume that each transition step takes exactly one time-unit. Configurations (in) *Conf* are pairs consisting of a process and a constraint in $\mathcal{C}$ representing the common *store*. The transition relation $\longrightarrow \subseteq Conf \times Conf$ is the least relation satisfying the rules **R1-R10** in Table 1 and characterizes the (temporal) evolution of the system. So, $\langle A, c \rangle \longrightarrow \langle B, d \rangle$ means that if at time t we have the process A and the store c then at time $t + 1$ we have the process B and the store d.

In order to represent successful termination in Table 1 we introduce the auxiliary agent $\mathbf{stop}$: it cannot make any transition. Rule **R1** shows that the agent $\mathbf{tell}(c)$ adds c to the store d (note that the updated store $c \sqcup d$ will be visible only starting from the next time instant). According to rule $\mathbf{R2}$ the guarded choice operator gives rise to global non-determinism, since the external environment can affect the choice. The rules $\mathbf{R3}$-$\mathbf{R6}$ show that the agent **now** c

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

Table 1. The transition system for $tccp$.

then A **else** B behaves as A or B depending on the (instantaneous) evaluation of the guard c[1]. Rules **R7** and **R8** model the parallel composition operator in terms of *maximal parallelism*. The agent $\exists xA$ behaves like A, with x considered *local* to A. To describe locality in rule **R9** the syntax has been extended by an agent $\exists^d xA$ where d is a local store of A containing information on x which is hidden in the external store. Initially the local store is empty, i.e. $\exists xA = \exists^{true} xA$. Rule **R10** treats the case of a procedure call when the actual parameter equals the formal parameter. We do not need more rules since here and in the following we assume that, for every procedure call p(y) appearing in a process D.A, if the original declaration for p in D is p(x):- A then D contains also the declaration p(y):- $\exists^{d_{xy}}$ xA.

Using the transition system described by (the rules in) Table 1 we can define a notion of (compositional) observables which associates with an agent a set of timed reactive sequences of the form $\langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle\langle d, d\rangle$ where a pair of constraints $\langle c_i, d_i\rangle$ represents a computation step performed by the agent at time i: Intuitively, the agent transforms the global store from c_i to d_i or, in other words, c_i is the assumption on the external environment while d_i is the contribution of the agent itself. The last pair denotes a "stuttering step" in which no further information can be produced by the agent, thus indicating that a "resting point" has been reached. Since in *tccp* computations the store evolves monotonically, it is natural to assume that reactive sequences are monotonically increasing, so we assume that each timed reactive sequence $\langle c_1, d_1\rangle \cdots \langle c_{n-1}, d_{n-1}\rangle\langle c_n, c_n\rangle$ satisfies the following condition: $c_i \leq d_i$ and $d_j \leq c_{j-1}$, for any $i \in [1, n-1]$ and $j \in [2, n]$. Since the constraints arising from computation steps are finite, we also assume that a reactive sequence contains only finite constraints. The set of all reactive sequences is denoted by $\mathcal{S}$ and its typical elements

[1] As discussed in [**?**], the evaluation of the guard in this case need to be instantaneous to be able to express such a construct as a time-out in the language.

by $s, s_1 \ldots$, while sets of reactive sequences are denoted by $S, S_1 \ldots$ and ε indicates the empty reactive sequence. Furthermore, $\cdot$ denotes the operator which concatenates sequences. Operationally the reactive sequences of an agent are generated as follows.

Definition 2.3 We define inductively the semantics $R \in Agent \to \mathcal{P}(\mathcal{S})$ by

$$\begin{array}{rl} R(\mathsf{A}) = & \{\langle c, d\rangle \cdot w \mid \langle \mathsf{A}, c\rangle \to \langle \mathsf{B}, d\rangle \text{ and } w \in R(\mathsf{B})\} \\ & \cup \\ & \{\langle c, c\rangle \cdot w \mid \langle \mathsf{A}, c\rangle \not\to \text{ and } w \in R(\mathsf{A}) \cup \{\varepsilon\}\}. \end{array}$$

Formally R is defined as the least fixed-point of the corresponding operator $\Phi \in (Agent \to \mathcal{P}(\mathcal{S})) \to (Agent \to \mathcal{P}(\mathcal{S}))$ whose definition $\Phi(I)(\mathsf{A})$ is obtained from the previous one by substituting $R(X)$ for $I(X)$ ($X \in \{A, B\}$). The set of interpretations $Agent \to \mathcal{P}(\mathcal{S})$ is easily seen to be a cpo, with the ordering of (point-wise extended) set-inclusion. It is also straightforward to check that Φ is continuous. According to the previous definition, the semantics $R(A)$ associates to a process A the set of all finite, maximal, reactive sequences obtained from the transition system by performing at each time instant an arbitrary assumption on the contribution of the external world . More abstract notion of observables (e.g. input/output pairs) can be extracted from reactive sequences (see [4]).

3 A calculus for tccp

In this section we introduce a temporal logic of knowledge and belief for reasoning about *tccp* programs. We first define temporal formulas and the related notions of truth and validity in terms of timed reactive sequences. Then we introduce the correctness assertions that we consider and a corresponding proof system.

3.1 Temporal logic

Our temporal logic is based on *epistemic* formulas of the form $K(c)$ and $B(c)$, where c is a constraint of the given underlying constraint system. The modality B expresses the "believes" of a process, that is, $B(c)$ holds if the process assumes the information represented by c is produced by the environment. On the other hand, the modality K represents the (private) knowledge of a process, that is, $K(c)$ holds if the information represented by c is "known" by the process itself. More precisely, these epistemic formulas will be interpreted with respect to a *reaction* which consists of pair of constraints $\langle c, d\rangle$, where, as previously mentioned, c represents the input of the external environment and d what is produced by the process itself. Formally we have the following truth definition of epistemic formulas.

Definition 3.1 Let $\mathcal{C}$ be a constraint system and $c, d, e \in \mathcal{C}$. Then we define $\langle c, d\rangle \models B(e)$ iff $e \leq c$ and $\langle c, d\rangle \models K(e)$ iff $e \leq d$.

An atomic formula in our temporal logic is an epistemic formula as described above. Compound formulas are constructed from these atomic formulas by using the (usual) logical operators of negation, conjunction and (existential) quantification (over the variables of the underlying constraint system) and the temporal operators $\bigcirc$ (the next operator) and $\mathcal{U}$ (the until operator).

Definition 3.2 [Temporal formulas] Given an underlying constraint system with set of constraints $\mathcal{C}$, formulas of the temporal logic are defined by

$$\begin{array}{rl} \phi ::= & K(c) \mid B(c) \mid \neg\phi \mid \phi \wedge \psi \mid \exists x \phi \mid \\ & \bigcirc\phi \mid \phi \,\mathcal{U}\, \psi \end{array}$$

In the sequel we assume that the temporal operators have binding priority over the propositional connectives. We introduce the following abbreviations: $\Diamond\phi$ for $true \;\mathcal{U}\; \phi$ and $\Box\phi$ for $\neg\Diamond\neg\phi$. We also use $\phi \vee \psi$ as a shorthand for $\neg(\neg\phi \wedge \neg\psi)$ and $\phi \to \psi$ as a shorthand for $\neg\phi \vee \psi$.

The truth of temporal formulas is defined with respect to a a timed reactive sequence $v_1 \cdots v_n$, where each view $v_i = \langle c_i, d_i\rangle$ describes the local knowledge and belief of the process at time i. Moreover we assume that time does not stop, so actually a finite sequence $v_1 \cdots v_n$ represents the infinite sequence $v_1 \cdots v_n v_n v_n \cdots$, with the last state repeated infinitely many times. Intuitively then $\bigcirc\phi$ holds if ϕ holds in the next time-instant and $\phi \,\mathcal{U}\, \psi$ holds if there exists a future moment (possibly the present) in which ϕ holds and until then ψ holds. In order to define formally the truth of a temporal formula we introduce the following notions: $s < s'$ if s is a proper suffix of s' ($s \leq s'$ if $s < s'$ or $s = s'$). Furthermore, for $s = v_1 \cdots v_n$, we define $next(s) = v_2 \cdots v_n$ (as a particular case, we have that $next(v) = v$) and $first(s) = v_1$.

Definition 3.3 Let s be a timed reactive sequence and ϕ be a temporal formula. Then we define $s \models \phi$ by:

$$\begin{array}{lll} s \models K(c) & \text{if} & first(s) \models K(c) \\ s \models B(c) & \text{if} & first(s) \models B(c) \\ s \models \neg\phi & \text{if} & s \not\models \phi \\ s \models \phi_1 \wedge \phi_2 & \text{if} & s \models \phi_1 \text{ and } s \models \phi_2 \\ s \models \exists x \phi & \text{if} & s' \models \phi, \text{ for some } s' \text{ s.t. } \exists_x s = \exists_x s' \\ s \models \bigcirc\phi & \text{if} & next(s) \models \phi \\ s \models \phi \,\mathcal{U}\, \psi & \text{if} & \text{for some } s' \leq s,\ s' \models \psi \text{ and} \\ & & \quad \text{for all } s' < s'' \leq s,\ s'' \models \phi. \end{array}$$

where, for $s = \langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle$, we define $\exists_x s = \langle \exists_x c_1, \exists_x d_1\rangle \cdots \langle \exists_x c_n, \exists_x d_n\rangle$.

Definition 3.4 A formula ϕ is valid, notation $\models \phi$, iff $s \models \phi$ for every timed reactive sequence s.

Previous definition implies for example the validity of the formula $\bigcirc true$ which expresses that time does not stop. Moreover we have the validity of the usual temporal tautologies. Monotonicity of the operators K and B wrt the entailment relation of the underlying constraint system is expressed by the following rules:

$$\frac{c \leq d}{B(d) \rightarrow B(c)} \text{ and } \frac{c \leq d}{K(d) \rightarrow K(c)}.$$

As discussed in the previous section, we assume that reactive sequences are monotonically increasing, that is, for each sequence $\langle c_1, d_1\rangle \cdots \langle c_{n-1}, d_{n-1}\rangle\langle c_n, c_n\rangle$ we assume that $c_i \leq d_i$ and $d_i \leq c_{i-1}$ for each $i \in [1, n-1]$. Then previous definition and Definition 3.1 imply the validity of the following formulas

$$B(c) \rightarrow \Box B(c) \text{ and } K(c) \rightarrow \Box K(c)$$

These formulas express the persistence of knowledge and believes, respectively. The relation between the epistemic operators is logically described by the laws

$$B(c) \rightarrow K(c) \text{ and } K(c) \rightarrow \bigcirc B(c),$$

that is, what is believed is also known and what is known at the current state is believed in the next state. The validity of these formulas depends on the monotonicity of the language that we are considering: In an imperative language, where information can be deleted, clearly these formulas are no longer valid.

3.2 The proof-system

We introduce now a proof-system for correctness of *tccp* programs. We first define formally the correctness assertions and their validity.

Definition 3.5 Correctness assertions are formulas of the form $A \ sat \ \phi$, where A is a *tccp* process and ϕ is a temporal formula. The validity of an assertion $A \ sat \ \phi$, denoted by $\models A \ sat \ \phi$, is defined as follows: $\models A \ sat \ \phi$ iff $s \models \phi$, for all $s \in R(A)$.

Thus $A \ sat \ \phi$ is valid if every (finite, maximal) computation of A satisfies ϕ. Table 2 presents the proof-system.

Axiom **T1** states that every computation of $\mathbf{tell}(c)$ satisfies $K(c)$, that is, after the execution of the tell-operation the process knows c. In rule **T2** B_i stands for $B(c_i)$. Given that A_i satisfies ϕ_i, rule **T2** allows the derivation of the specification for $\Sigma_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$, which expresses that either eventually c_i is believed and, consequently, ϕ_i holds in the *next* time-instant (since the evaluation of the ask takes one time-unit), or none of the guards is ever satisfied. Rule **T3** simply states that if A satisfies ϕ and B satisfies ψ then every computation of **now** c **then** A **else** B satisfies either ϕ or ψ, depending on the fact that c is believed or not. Hiding of a local variable x is axiomatized in rule **T4** by existentially quantifying x in ϕ. Rule **T5** simply models parallel composition in terms of conjunction. Rule **T6** describes recursion in the usual manner (see also [3]) and as in Table 1, x is assumed to be both the formal and the actual parameter (here $\vdash$ denotes derivability within the proof system). Rule **T7** allows to weaken the specification.

3.3 Soundness and completeness

We denote by $\vdash A \ sat \ \phi$ the derivability of the correctness assertion $A \ sat \ \phi$ in the proof system introduced in the previous section. The following theorem states the soundness of this proof system.

Theorem 3.6 *Let A be a tccp process. If $\vdash A \ sat \ \phi$ then $\models A \ sat \ \phi$, for every correctness assertion $A \ sat \ \phi$.*

At the heart of this theorem lies the compositionality of $R(A)$ which is proved in [4]: Intuitively, for each syntactic construct of the language we can define a semantic operator (acting on sets of reactive sequences) which reflects the operational behaviour of the syntactic operator. For example, given two sets of reactive sequences S_1, S_2, the semantic parallel operator $S_1 \tilde{\|} S_2$ is defined as the point-wise extension to sets of the following (commutative and associative) partial operator defined on sequences:

$$\langle c_1, d_1\rangle \cdots \langle c_n, d_n\rangle\langle d, d\rangle \tilde{\|} \langle c_1, e_1\rangle \cdots \langle c_n, e_n\rangle\langle d, d\rangle = \langle c_1, d_1 \sqcup e_1\rangle \cdots \langle c_n, d_n \sqcup e_n\rangle\langle d, d\rangle.$$

where we require that the two arguments of the operator agree at each point of time with respect to the contribution of the environment (the c_i's) and that they have the same length (in all other cases $\tilde{\|}$ is assumed being undefined).

Given the compositionality of the semantics R, soundness can be proved by induction on the length of the derivation. A crucial property of the proof system needed in the soundness proof (for the rule of parallel composition) is the following: if $\vdash A \ sat \ \phi$ and s' is a reactive sequence which is obtained from a reactive sequence $s \in R(A)$ by replacing some 'stutter' reactions $\langle c, c\rangle$ by $\langle c, d\rangle$, for some $c \leq d$, then $s' \models \phi$. In other words, correctness assertions which are derivable do *not* fully specify stutter reactions. Note that the previous property does not hold when considering generic formulas. For example, one has that $\langle true, true\rangle \models \neg K(a)$ while $\langle true, a\rangle \not\models \neg K(a)$. Because of underspecification of stutter reactions we do not have completeness in the sense that every valid correctness assertion is derivable.

T1 $\mathbf{tell(c)}\ sat\ K(c)$

T2 $$\frac{A_i\ sat\ \phi_i, \forall i \in [1,n]}{\sum_{i=1}^{n} ask(c_i) \rightarrow A_i\ sat\ \bigvee_{i=1}^{n}((\bigwedge_{j=1}^{n}\neg B_j)\ \mathcal{U}\ (B_i \wedge \bigcirc\phi_i)) \vee \Box \bigwedge_{j=1}^{n}\neg B_j}$$

T3 $$\frac{A\ sat\ \phi \quad B\ sat\ \psi}{\textbf{now}\ c\ \textbf{then}\ A\ \textbf{else}\ B\ sat\ (B(c) \wedge \phi) \vee (\neg B(c) \wedge \psi)}$$

T4 $$\frac{A\ sat\ \phi}{\exists_x A\ sat\ \exists x(\phi)}$$

T5 $$\frac{A\ sat\ \phi \quad B\ sat\ \psi}{A \parallel B\ sat\ \phi \wedge \psi}$$

T6 $$\frac{p(x)\ sat\ \phi \vdash A\ sat\ \phi}{p(x)\ sat\ \phi}$$ $p(x)$ declared as A

T7 $$\frac{A\ sat\ \phi \quad \models \phi \rightarrow \psi}{A\ sat\ \psi}$$

Table 2. The system TL for $tccp$.

Completeness can be obtained by extending the logic in order to allow for a more precise representation of stutter reactions. If we allow in the logic constraint variables $p, q \ldots$ (and quantification over them) a stutter reaction can be fully specified by the formula $\forall p(K(p) \leftrightarrow B(p))$ which states that what is believed is known and vice versa. With such an extension, for example, axiom $T1$ can be strengthened as follows: $\mathbf{tell(c)}\ sat\ K(c) \wedge \Box\forall p(K(p) \leftrightarrow (B(p) \vee p \leq c))$ where $p \leq c$ is an atomic formula which 'imports' information about the underlying constraint system. However, such an extension requires a major revision of the proof system since the simple version of rule $T5$ that we used would be no longer sound (a suitable notion of quantification over predicate variables is required). The development of a complete calculus is deferred to the extended version of this paper.

4 Related and future work

We introduced a temporal logic for reasoning about the correctness of a timed extension of ccp and we proved the soundness of a related proof system.

Recently, a logic for a different timed extension of ccp, called ntcc, has been presented in [13]. The language ntcc is a non deterministic extension of the timed ccp language defined in [15]. Its computational model, and therefore the underlying logic, are rather different from those that we considered, since both ntcc and the language defined in [15] follow the ESTEREL model of computation consisting of "bursts of activity": In each phase a ccp process is executed to produce a response to an input provided by the environment. Thus, each time interval is identified with the time needed for a ccp process to terminate a computation (suitable syntactic restrictions on recursion ensure that the program is always terminating). The programmer has to transfer explicitly the all information from a time instant to the next one by using special primitives, since at the end of a time interval all the constraints accumulated and all the processes suspended are discarded. These assumptions allow to obtain an elegant semantic model consisting of sequences of sets of resting points (each set describing the behaviour at a time instant).

On the other hand, the tccp language that we consider has a different notion of time, since each time-unit is identified with the time needed for the underlying constraint system to accumulate the tell's and to answer the ask's issued at each computation step by the processes of the system. This assumption allow us to maintain the essential features of ccp computations: No restriction on recursion is needed (since at each time instant there is a finite number of parallel agents) and no explicit transfer of information across time boundaries is required.These differences affects also the expressive power of the language (see [4] for a detailed discussion). Since the store grows monotonically, some syntactic restrictions are needed also in tccp in order to obtain bounded response time, that is, to be able to statically determine the maximal length of each time-unit (see [4]).

From a logical point of view, as shown in [3] the set of resting points of a ccp program characterizes essentially the strongest post condition of the program (the characterization however is exact only for a certain class of programs). In [13] this logical view is integrated with (linear) temporal logic constructs which are interpreted in terms of sequences of sets of resting points, thus taking into account the temporal evolution of the system. Since the resting points provide a compositional model (describing the final results of computations), in this approach there is no need for a semantic and logical representation of "assumptions". On the other hand such a need arise when one wants to describe the input/output behaviour of a process, which for generic (non deterministic) processes cannot be obtained from the resting points. Since tccp maintains essentially the ccp computational model, at each time instant rather than a set of final results (i.e. a set of resting points) we have an input/ouput behaviour corresponding to the interaction of the environment, which provides the input, with the process, which produces the output. This is reflected in the semantic structures that we use and in the logic we have defined, which could be seen a logic for describing timed reactive sequences.

Related to the present paper is also [8], where tcc specifications are represented in terms of graph structures in order to apply model checking techniques. A finite interval of time (introduced by the user) is considered in order to obtain a finite behaviour of the tcc program, thus allowing the application of existing model checking algorithms.

Future work concerns first of all the development of a sound and complete axiomatization of the temporal logic introduced in this paper and an investigation into decision procedures. Since *reactive sequences* have been used also in the semantics of several other languages, including dataflow and imperative ones [11, 5, 7], we plan also to consider extensions of our logic to deal with these different languages.

References

[1] J. Baeten and J. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2): 142-188, 1991.

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

[3] F.S. de Boer, M. Gabbrielli, E. Marchiori and C. Palamidessi. Proving Concurrent Constraint Programs Correct. *Transactions on Programming Languages and Systems (TOPLAS)*, 19(5): 685-725. ACM Press, 1997.

[4] F.S. de Boer, M. Gabbrielli and M.C. Meo. A Timed CCP Language. *Information and Computation*, 161, 2000.

[5] F.S. de Boer, J.N. Kok, C. Palamidessi, and J.J.M.M. Rutten. The failure of failures in a paradigm for asynchronous communication. In J.C.M. Baeten and J.F. Groote, editors, *Proceedings of CONCUR'91*, vol. 527 of *LNCS*, pages 111–126. Springer-Verlag, 1991.

[6] F.S. de Boer and C. Palamidessi. A Fully Abstract Model for Concurrent Constraint Programming. In S. Abramsky and T.S.E. Maibaum, editors, *Proc. of TAPSOFT/CAAP*, vol. 493 of *LNCS*, pages 296–319. Springer-Verlag, 1991.

[7] S. Brookes. A fully abstract semantics of a shared variable parallel language. In *Proc. Eighth IEEE Symposium on Logic In Computer Science*. IEEE Computer Society Press, 1993.

[8] M. Falaschi, A. Policriti, A. Villanueva. Modeling Concurrent systems specified in a Temporal Concurrent Constraint language. in *Proc. AGP'2000*. 2000.

[9] M. Fisher. An introduction to Executable Temporal Logics. *Knowledge Engineering Review*, 6(1): 43-56, 1996.

[10] M. Hennessy and T. Regan. A temporal process algebra. *Information and Computation*, 117: 221-239, 1995.

[11] B. Jonsson. A model and a proof system for asynchronous processes. In *Proc. of the 4th ACM Symp. on Principles of Distributed Computing*, pages 49–58. ACM Press, 1985.

[12] Z. Manna and A. Pnueli. *The temporal logic of reactive systems*. Springer-Verlag, 1991.

[13] C. Palamidessi and F.D. Valencia. A Concurrent Constraint Calculus for timed systems. *Draft*, 2000.

[14] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of POPL*, pages 232–245. ACM Press, 1990.

[15] V.A. Saraswat, R. Jagadeesan, and V. Gupta Foundations of Timed Concurrent Constraint Programming . In *Proc. of LICS 94*, 1994.

[16] V.A. Saraswat, R. Jagadeesan, and V. Gupta Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5-6):475–520, 1996.