Proving Correctness of Timed Concurrent Constraint Programs

Frank S. de Boer[1], Maurizio Gabbrielli[2], and Maria Chiara Meo[3]

[1] Universiteit Utrecht,
Department of Computer Science,
Padualaan 14, De Uithof, 3584 CH, Utrecht, The Netherlands
frankb@cs.ruu.nl
[2] Università di Bologna
Dipartimento di Scienze dell'Informazione
Mura A. Zamboni 7, 40127 Bologna, Italy
gabbri@dimi.uniud.it
[3] Università di Chieti
Dipartimento di Scienze
Viale Pindaro 42, 65127 Pescara, Italy
meo@univaq.it

Abstract. A temporal logic is presented for reasoning about the correctness of timed concurrent constraint programs. The logic is based on modalities which allow one to specify what a process produces as a reaction to what its environment inputs. These modalities provide an assumption/commitment style of specification which allows a sound and complete compositional axiomatization of the reactive behavior of timed concurrent constraint programs.

Keywords: Concurrency, constraints, real-time programming, temporal logic.

1 Introduction

Many "real-life" computer applications maintain some ongoing interaction with external physical processes and involve time-critical aspects. Characteristic of such applications, usually called real-time embedded systems, is the specification of timing constraints such as, for example, that an input is required within a bounded period of time. Typical examples of such systems are process controllers and signal processing systems.

In [5] *tccp*, a timed extension of the pure formalism of concurrent constraint programming([24]), is introduced. This extension is based on the hypothesis of *bounded asynchrony* (as introduced in [26]): Computation takes a bounded period of time rather than being instantaneous as in the concurrent synchronous languages `ESTEREL` [3], `LUSTRE` [15], `SIGNAL` [19] and Statecharts [16]. Time itself is measured by a discrete global clock, i.e, the internal clock of the *tccp* process. In [5] we also introduced *timed reactive sequences* which describe at each moment in time the reaction of a tccp process to the input of the external environment. Formally, such a reaction is 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 in response to the input c (such a response includes always the input because of the monotonicity of ccp computations).

In this paper we introduce a temporal logic for describing and reasoning about timed reactive sequences. The basic assertions of the temporal logic describe the reactions of such a sequence in terms of *modalities* which express either what a process *assumes* about the inputs of the environment and what a process *commits* to, i.e., has itself produced at one time-instant. These modalities thus provide a kind of assumption/commitment style of specification of the reactive behavior of a process. The main result of this paper is a sound and complete compositional proof system for reasoning about the correctness of *tccp* programs as specified by formulas in this temporal logic.

The remainder of this paper is organized as follows. In the next section we introduce the language *tccp* and its operational semantics. In Section 3 we introduce the temporal logic and the compositional proof system. in Section 4 we briefly discuss soundness and completeness of the proof system. Section 5 concludes by discussing related work and indicating future research.

2 The Programming Language

In this section we first define the *tccp* language and then we define formally its operational semantics by using a transition system.

Since the starting point is ccp, we introduce first some basic notions related to this programming paradigm. We refer to [25,27] for more details. The ccp languages are defined parametrically wrt to a given *constraint system*. The notion of constraint system has been formalized in [25] following Scott's treatment of information systems. Here we only consider the resulting structure.

Definition 1. *A constraint system is a complete algebraic lattice $\langle \mathcal{C}, \leq, \sqcup, true, false\rangle$ where $\sqcup$ is the lub operation, and true, false are the least and the greatest elements of $\mathcal{C}$, respectively.*

Following the standard terminology and notation, instead of $\leq$ we will refer to its inverse relation, denoted by $\vdash$ and called *entailment*. Formally, $\forall c, d \in C.\ \ c \vdash d \ \Leftrightarrow\ d \leq c$. In order to treat the hiding operator of the language a general notion of existential quantifier is introduced which is formalized in terms of cylindric algebras [17]. Moreover, in order to model parameter passing, *diagonal elements* [17] are added to the primitive constraints. This leads to the concept of a *cylindric constraint system*. In the following, we assume given a (denumerable) set of variables *Var* with typical elements $x, y, z, \ldots$.

Definition 2. *Let $\langle \mathcal{C}, \leq, \sqcup, true, false\rangle$ be a constraint system. Assume that for each $x \in$ Var a function $\exists_x : \mathcal{C} \rightarrow \mathcal{C}$ is defined such that for any $c, d \in \mathcal{C}$:*

(i) $c \vdash \exists_x(c)$, (ii) *if* $c \vdash d$ *then* $\exists_x(c) \vdash \exists_x(d)$,
(iii) $\exists_x(c \sqcup \exists_x(d)) = \exists_x(c) \sqcup \exists_x(d)$, (iv) $\exists_x(\exists_y(c)) = \exists_y(\exists_x(c))$.

Moreover assume that for x, y ranging in Var, $\mathcal{C}$ contains the constraints d_{xy} (so called diagonal elements) which satisfy the following axioms:

(v) *true* $\vdash d_{xx}$, (vi) *if* $z \neq x, y$ *then* $d_{xy} = \exists_z(d_{xz} \sqcup d_{zy})$,
(vii) *if* $x \neq y$ *then* $d_{xy} \sqcup \exists_x(c \sqcup d_{xy}) \vdash c$.

Then $\mathbf{C} = \langle \mathcal{C}, \leq, \sqcup, \mathit{true}, \mathit{false}, \mathit{Var}, \exists_x, d_{xy} \rangle$ *is a cylindric constraint system.*

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 (so, for instance, $\exists_x c \sqcup d$ stands for $\exists_x(c) \sqcup d$).

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. Consider for example the case of a bank teller machine. Once a card is accepted and its identification number has been checked, the machine asks the authorization of the bank to release the requested money. If the authorization does not arrive within a reasonable amount of time, then the card should be given back to the customer. A timed language should then allow us to specify that, in case a given time bound is exceeded (i.e. a *time-out* occurs), the wait is interrupted and an alternative action is 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). For example, according to a typical pattern, A is the process controlling the normal activity of some physical device, the event indicates some abnormal situation and B is the exception handler.

In order to be able to specify these timing constraints 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. Thus essentially in our model all parallel agents are synchronized by 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 [5,26] 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 3 (*tccp* Language [5]). *Assuming a given cylindric constraint system* $\mathbf{C}$ *the syntax of* agents *is given by the following grammar:*

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

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 $\sum_{i=1}^{n} \mathbf{ask}(c_i) \rightarrow A_i$, 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 the next subsection we describe formally the operational semantics of *tccp*. In order to simplify the notation, in the following we will omit the $\sum_{i=1}^{n}$ whenever $n = 1$ and we will use *tell(c)* $\rightarrow A$ as a shorthand for *tell*$(c) \parallel (\mathbf{ask}(\mathit{true}) \rightarrow A)$. In the following we also assume guarded recursion, that is we assume that each procedure call is in the scope of an **ask** construct. This assumption, which does not limit the expressive power of the language, is needed to ensure a proper definition of the operational semantics.

2.1 Operational Semantics

The operational model of *tccp* can be formally described by a transition system $T = (\mathit{Conf}, \longrightarrow)$ 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 $\longrightarrow \subseteq \mathit{Conf} \times \mathit{Conf}$ is the least relation satisfying the rules **R1-R11** 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. As usual, $\langle A, c\rangle \not\longrightarrow$ means that there exist no transitions for the configuration $\langle A, c\rangle$.

Let us now briefly discuss the rules in Table 1. In order to represent successful termination we introduce the auxiliary agent **stop**: it cannot make any transition. Rule **R1** shows that we are considering here the so called "eventual" tell: The agent $\mathbf{tell}(c)$ adds c to the store d without checking for consistency of $c \sqcup d$ and then stops. Note that the updated store $c \sqcup d$ will be visible only starting from the next time instant since each transition step involves exactly one time-unit. According to rule **R2** the guarded choice operator gives rise to global non-determinism: The external environment can affect the choice since $\mathbf{ask}(c_j)$ is enabled at time t (and A_j is started at time $t+1$) iff the store d

Table 1. The transition system for *tccp*.

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* $d \vdash c_j$
R3	$\dfrac{\langle A, d\rangle \longrightarrow \langle A', d'\rangle}{\langle \mathbf{now}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, d\rangle \longrightarrow \langle A', d'\rangle}$	$d \vdash c$
R4	$\dfrac{\langle A, d\rangle \not\longrightarrow}{\langle \mathbf{now}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, d\rangle \longrightarrow \langle A, d\rangle}$	$d \vdash c$
R5	$\dfrac{\langle B, d\rangle \longrightarrow \langle B', d'\rangle}{\langle \mathbf{now}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, d\rangle \longrightarrow \langle B', d'\rangle}$	$d \not\vdash c$
R6	$\dfrac{\langle B, d\rangle \not\longrightarrow}{\langle \mathbf{now}\ c\ \mathbf{then}\ A\ \mathbf{else}\ B, d\rangle \longrightarrow \langle B, d\rangle}$	$d \not\vdash c$
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	$\dfrac{\langle A, c\rangle \longrightarrow \langle B, d\rangle}{\langle p(x), c\rangle \longrightarrow \langle B, d\rangle}$	$p(x) : -A \in D$

entails c_j, and d can be modified by other agents. The rules **R3-R6** show that the agent **now** c **then** A **else** B behaves as A or B depending on the fact that c is or is not entailed by the store. Differently from the case of the ask, here the evaluation of the guard is instantaneous: If $\langle A, d\rangle$ ($\langle B, d\rangle$) can make a transition at time t and c is (is not) entailed by the store d, then the agent **now** c **then** A **else** B can make the same transition at time t[1]. Moreover, observe that in any case the control is passed either to A (if c is entailed by the current store d) or to B (in case d does not entail c). Rules **R7** and **R8** model the parallel composition operator in terms of *maximal parallelism*: The agent $A \parallel B$ executes in one time-unit all the initial enabled actions of A and B. Thus, for example, the agent $A :\ (\mathbf{ask}(c) \rightarrow stop) \parallel (\mathbf{tell}(c) \rightarrow stop)$ evaluated in the store c will (successfully) terminate in one time-unit, while the same agent in the empty

[1] As discussed in [5], the evaluation of the guard needs to be instantaneous to be able to express in the *tccp* language such a construct as a time-out.

store will take two time-units to terminate. The agent $\exists x A$ behaves like A, with x considered *local* to A, i.e. the information on x provided by the external environment is hidden to A and, conversely, the information on x produced locally by A is hidden to the external world. To describe locality in rule **R9** the syntax has been extended by an agent $\exists^d x A$ 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. $\exists x A = \exists^{true} x A$.

Rule **R10** treats the case of a procedure call when the actual parameter equals the formal parameter. We do not need more rules since, for the sake of simplicity, here and in the following we assume that the set D of procedure declarations is closed wrt parameter names: That is, for every procedure call *p(y)* appearing in a process $D.A$ we assume that if the original declaration for p in D is *p(x):-* A then D contains also the declaration *p(y):-* $\exists^{d_{xy}} x A$[2].

Using the transition system described by (the rules in) Table 1 we can now define our notion of 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 reaction of the given 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 (which includes always the assumption). 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 the basic actions of *tccp* are monotonic and we can also model a new input of the external environment by a corresponding tell operation, it is natural to assume that reactive sequences are monotonically increasing. So in the following we will 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: $d_i \vdash c_i$ and $c_j \vdash d_{j-1}$, for any $i \in [1, n-1]$ and $j \in [2, n]$. Since the constraints arising from the reactions are finite, we also assume that a reactive sequence contains only finite constraints[3].

The set of all reactive sequences is denoted by $\mathcal{S}$ and its typical elements 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 4. *We define inductively the semantics $R \in Agent \to \mathcal{P}(\mathcal{S})$ by*

$$\begin{aligned} R(A) = &\{\langle c, d\rangle \cdot w \in \mathcal{S} \mid \langle A, c\rangle \to \langle B, d\rangle \ \textit{and}\ w \in R(B)\} \\ &\cup \\ &\{\langle c, c\rangle \cdot w \in \mathcal{S} \mid \langle A, c\rangle \not\to \ \textit{and}\ w \in R(A) \cup \{\varepsilon\}\}. \end{aligned}$$

[2] Here the (original) formal parameter is identified as a local alias of the actual parameter. Alternatively, we could have introduced a new rule treating explicitly this case, as it was in the original ccp papers.

[3] Note that here we implicitly assume that if c is a finite element then also $\exists_x c$ is finite.

Note that $R(A)$ is defined as the union of the set of all reactive sequences which start with a reaction of A and the set of all reactive sequences which start with a stuttering step of A. In fact, when an agent is blocked, i.e., it cannot react to the input of the environment, a stuttering step is generated. After such a stuttering step the computation can either continue with the further evaluation of A (possibly generating more stuttering steps) or it can terminate, as a "resting point" has been reached. These two case are reflected in the second part of the definition of $R(A)$ by the two conditions $w \in R(A)$ and $w \in \{\varepsilon\}$, respectively. Note also that, since the stop agent used in the transition system cannot make any move, an arbitrary (finite) sequence of stuttering steps is always appended to each reactive sequence.

Formally R is defined as the least fixed-point of the corresponding operator $\Phi \in (Agent \rightarrow \mathcal{P}(\mathcal{S})) \rightarrow Agent \rightarrow \mathcal{P}(\mathcal{S})$ defined by

$$\begin{aligned}\Phi(I)(A) = \; & \{\langle c,d\rangle \cdot w \in \mathcal{S} \mid \langle A,c\rangle \rightarrow \langle B,d\rangle \text{ and } w \in I(B)\} \\ & \cup \\ & \{\langle c,c\rangle \cdot w \in \mathcal{S} \mid \langle A,c\rangle \not\rightarrow \text{ and } w \in I(A) \cup \{\varepsilon\}\}.\end{aligned}$$

The ordering on $Agent \rightarrow \mathcal{P}(\mathcal{S})$ is that of (point-wise extended) set-inclusion (it is straightforward to check that Φ is continuous).

3 A Calculus for tccp

In this section we introduce a temporal logic for reasoning about the reactive behavior of *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

Given a set M, with typical elements $X, Y, \ldots$, of monadic constraint predicate variables, our temporal logic is based on atomic formulas of the form $X(c)$, where c is a constraint of the given underlying constraint system. The distinguished predicate I will be used to express the "assumptions" of a process about its inputs, that is, $I(c)$ holds if the process assumes the information represented by c is produced by its environment. On the other hand, the distinguished predicate O represents the output of a process, that is, $O(c)$ holds if the information represented by c is produced by the process itself (recall that the produced information includes always the input, as previously mentioned). More precisely, these formulas $I(c)$ and $O(c)$ will be interpreted with respect to a *reaction* which consists of a pair of constraints $\langle c, d\rangle$, where c represents the input of the external environment and d is the contribution of the process itself (as a reaction to the input c) which always contains c (i.e. such that $d \geq c$ holds).

An atomic formula in our temporal logic is a formula as described above or an atomic formula of the form $c \leq d$ which 'imports' information about the underlying constraint system, i.e., $c \leq d$ holds if $d \vdash c$. Compound formulas are constructed from these atomic formulas by using the (usual) logical operators of negation, conjunction and (existential) quantification and the temporal operators $\bigcirc$ (the next operator) and $\mathcal{U}$ (the until operator). We have the following three different kinds of quantification:

- quantification over the variables $x, y, \ldots$ of the underlying constraint system;
- quantification over the constraints $c, d, \ldots$ themselves;
- quantification over the monadic constraint predicate variables $X, Y, \ldots$.

Variables $p, q, \ldots$ will range over the constraints. We will use $V, W, \ldots$, to denote a variable x of the underlying constraint system, a constraint variable p or a constraint predicate X.

Definition 5 (Temporal formulas). *Given an underlying constraint system with set of constraints $\mathcal{C}$, formulas of the temporal logic are defined by*

$$\phi ::= p \leq q \mid X(c) \mid \neg\phi \mid \phi \wedge \psi \mid \exists V \phi \mid \bigcirc \phi \mid \phi \, \mathcal{U} \, \psi$$

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* $\mathcal{U}$ ϕ 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 $\phi \rightarrow \psi$ as a shorthand for $\neg\phi \vee \psi$. Finally, $c = d$ stands for $c \leq d \wedge d \leq c$.

Definition 6. *Given an underlying constraint system with set of constraints $\mathcal{C}$, the truth of an atomic formula $X(c)$ is defined with respect to a predicate assignment $v \in M \rightarrow C$ which assigns to each monadic predicate X a constraint. We define*

$$v \models X(c) \text{ if } v(X) \vdash c.$$

Thus $X(c)$ holds if c is entailed by the constraint represented by X. In other words, a monadic constraint predicate X denotes a set $\{d \mid d \vdash c\}$ for some c. We restrict to constraint predicate assignments which are monotonic in the following sense: $v(O) \vdash v(I)$. In other words, the output of a process contains its input. The temporal operators are interpreted with respect to finite sequence $\rho = v_1, \ldots, v_n$ of constraint predicate assignments in the standard manner: $\bigcirc\phi$ 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 ψ holds and until then ϕ holds. We restrict to sequences $\rho = v_1, \ldots, v_n$ which are monotonic in the following sense: for $1 \leq i < n$, we have

- $v_{i+1}(X) \vdash v_i(X)$, for every predicate X;
- $v_{i+1}(I) \vdash v_i(O)$.

The latter condition requires that the input of a process contains its output at the previous time-instant. Note that these conditions corresponds with the monotonicity of reactive sequences as defined above. 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 element repeated infinitely many times.

In order to define formally the truth of a temporal formula we introduce the following notions: $\rho < \rho'$ if ρ is a proper suffix of ρ' ($\rho \leq \rho'$ if $\rho < \rho'$ or $\rho = \rho'$). Furthermore, for $\rho = v_1 \cdots v_n$, we denote by $\bigcirc\rho = v_2 \cdots v_n$ (as a particular case, we have that $\bigcirc v = v$) and $\rho_i = v_i$, $1 \leq i \leq n$. Given a variable x of the underlying constraint systems and a predicate assignment v we define the predicate assignment $\exists x v$ by $\exists x v(X) = \exists_x d$, where $d = v(X)$. Given a sequence $\rho = v_1, \ldots, v_n$, we denote by $\exists x \rho$ the sequence $\exists x v_1, \ldots, \exists x v_n$. Moreover, given a monadic constraint predicate X and a predicate assignment v we denote by $\exists X v$ the *restriction* of v to $M \setminus \{X\}$. Given a sequence $\rho = v_1, \ldots, v_n$, we denote by $\exists X \rho$ the sequence $\exists X v_1, \ldots, \exists X v_n$ Furthermore, by γ we denote a constraint assignment which assigns to each constraint variable p a constraint $\gamma(p)$. Finally, $\gamma\{c/p\}$ denotes the result of assigning in γ the constraint c to the variable p.

Definition 7. *Given a sequence of predicate assignments ρ, a constraint assignment γ and ϕ a temporal formula, we define $\rho \models_\gamma \phi$ by:*

$\rho \models_\gamma p \leq q$	if $\gamma(q) \vdash \gamma(p)$
$\rho \models_\gamma X(c)$	if $\rho_1 \models X(c)$
$\rho \models_\gamma \neg\phi$	if $\rho \not\models_\gamma \phi$
$\rho \models_\gamma \phi_1 \wedge \phi_2$	if $\rho \models_\gamma \phi_1$ and $\rho \models_\gamma \phi_2$
$\rho \models_\gamma \exists x \phi$	if $\rho' \models_\gamma \phi$, for some ρ' s.t. $\exists x \rho = \exists x \rho'$
$\rho \models_\gamma \exists X \phi$	if $\rho' \models_\gamma \phi$, for some ρ' s.t. $\exists X \rho = \exists X \rho'$
$\rho \models_\gamma \exists p \phi$	if $\rho \models_{\gamma'} \phi$, for some c s.t. $\gamma' = \gamma\{c/p\}$
$\rho \models_\gamma \bigcirc\phi$	if $\bigcirc\rho \models_\gamma \phi$
$\rho \models_\gamma \phi\, \mathcal{U}\, \psi$	if for some $\rho' \leq \rho$, $\rho' \models_\gamma \psi$ and for all $\rho' < \rho'' \leq \rho$, $\rho'' \models_\gamma \phi$.

Definition 8. *A formula ϕ is valid, notation $\models \phi$, iff $\rho \models_\gamma \phi$ for every sequence ρ of predicate assignments and constraint assignment γ.*

We have the validity of the usual temporal tautologies. Monotonicity of the constraint predicates wrt the entailment relation of the underlying constraint system is expressed by the formula

$$\forall p \forall q \forall X (p \leq q \rightarrow (X(q) \rightarrow X(p))).$$

Monotonicity of the constraint predicates wrt time implies the validity of the following formula

$$\forall p \forall X (X(p) \rightarrow \Box X(p)).$$

The relation between the distinguished constraint predicates I and O is logically described by the laws

$$\forall p (I(p) \rightarrow O(p)) \text{ and } \forall p (O(p) \rightarrow \bigcirc I(p)),$$

that is, the output of a process contains its input and is contained in the inputs of the next time-instant.

3.2 The Proof-System

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

Definition 9. *Correctness assertions are of the form A sat ϕ, where A is a* tccp *process and ϕ is a temporal formula. The validity of an assertion A sat ϕ, denoted by $\models A$ sat ϕ, is defined as follows*

$$\models A \ sat\ \phi \ \text{iff}\ \rho \models_\gamma \phi, \ \text{for all}\ \gamma \ \text{and}\ \rho \in R'(A),$$

where $R'(A) = \{v_1, \ldots, v_n \mid \langle v_1(I), v_1(O)\rangle \cdots \langle v_n(I), v_n(O)\rangle \in R(A)\}$.

Roughly, the correctness assertion A *sat* ϕ states that every sequence ρ of predicate assignments such that its 'projection' onto the distinguished predicates I and O generates a reactive sequence of A, satisfies the temporal formula ϕ.

Table 2. The system TL for *tccp*.

T1 $\mathbf{tell(c)}\ sat\ O(c) \wedge \forall p(O(p) \rightarrow \exists q(I(q) \wedge q \sqcup c = p)) \wedge \bigcirc\Box stut$

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

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

T4 $$\frac{A\ sat\ \phi}{\exists x A\ sat\ \exists x(\phi \wedge loc(x)) \wedge inv(x)}$$

T5 $$\frac{A\ sat\ \phi \quad B\ sat\ \psi}{A \parallel B\ sat\ \exists X, Y(\phi[X/O] \wedge \psi[Y/O] \wedge par(X,Y))}$$

T6 $$\frac{p(x)\ sat\ \phi \vdash_p A\ sat\ \phi}{p(x)\ sat\ \phi} \quad p(x) \text{ declared as } A$$

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

Table 2 presents the proof-system. Axiom **T1** states that the execution of **tell**(c) consists of the output of c (as described by $O(c)$) together with any possible input (as described by $I(q)$). Moreover, at every time-instant in the future no further output is generated, which is expressed by the formula

$$\forall p(O(p) \leftrightarrow I(p)),$$

which we abbreviate by *stut* (since it represents stuttering steps). In rule **T2** I_i stands for $I(c_i)$. 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 an input 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 an input or not. Hiding of a local variable x is axiomatized in rule **T4** by first existentially quantifying x in $\phi \wedge loc(x)$, where $loc(x)$ denotes the following formula which expresses that x is local, i.e., the inputs of the environment do not contain new information on x:

$$\forall p(\exists_x p \neq p \rightarrow (\neg I(p) \wedge \Box(\bigcirc I(p) \rightarrow \exists r(O(r) \wedge \exists_x p \sqcup r = p)))).$$

This formula literally states that the initial input does not contain information on x and that everywhere in the computation if in the next state an input contains information on x then this information is already contained by the previous output. Finally, the following formula $inv(x)$

$$\forall p \Box(\exists_x p \neq p \rightarrow (O(p) \rightarrow \exists r(I(r) \wedge \exists_x p \sqcup r = p)))$$

states that the process does not provide new information on the *global* variable x. Rule **T5** gives a compositional axiomatization of parallel composition. The 'fresh' constraint predicates X and Y are used to represent the outputs of A and B, respectively ($\phi[X/O]$ and $\psi[Y/O]$ denote the result of replacing O by X and Y). Additionally, the formula

$$\forall p \Box(O(p) \leftrightarrow (\exists q_1, q_2(X(q_1) \wedge Y(q_2) \wedge q_1 \sqcup q_2 = p))),$$

denoted by $par(X, Y)$, expresses that every output of $A \parallel B$ can be decomposed into outputs of A and B. Rule **T6**, where $\vdash_p$ denotes derivability within the proof system, describes recursion in the usual manner (see also [4]) and in this rule x is assumed to be both the formal and the actual parameter. We do not need more rules since, as previously mentioned, we may assume without loss of generality that the set D of procedure declarations is closed wrt parameter names. Rule **T7** allows to weaken the specification.

As an example of a sketch of a derivation consider the agent $\exists x A$ where

$$\begin{array}{ll} A :: & \mathbf{ask}(x = a) \rightarrow \mathbf{tell}(true) \\ & + \\ & \mathbf{ask}(true) \rightarrow \mathbf{tell}(y = b). \end{array}$$

(constraints are equations on the Herbrand universe). By **T1** and **T7** we derive

$$\mathbf{tell}(y = b)\ sat\ O(y = b) \quad and \quad \mathbf{tell}(true)\ sat\ O(true).$$

By **T2** and **T7** we subsequently derive $A\ sat\ I(x = a) \vee \bigcirc O(y = b)$ (note that $\neg I(true)$ is logically equivalent to *false* and *false* $\mathcal{U}\phi$ is equivalent to ϕ). Using rule **T4**, we derive the correctness assertion

$$\exists x A\ sat\ \exists x((I(x = a) \vee \bigcirc O(y = b)) \wedge loc(x)).$$

It is easy to see that $I(x = a) \wedge loc(x)$ implies *false*. So we have that $\exists x((I(x = a) \vee \bigcirc O(y = b)) \wedge loc(x))$ implies $\exists x(loc(x) \wedge \bigcirc O(y = b))$. Clearly this latter formula implies $\bigcirc O(y = b)$. Summarizing the above, we obtain a derivation of the correctness assertion $\exists x A\ sat\ \bigcirc O(y = b)$ which states that in every reactive sequence of $\exists x A$ the constraint $y = b$ is produced in the next (wrt the start of the sequence) time instant.

4 Soundness and Completeness

We denote by $\vdash_p A\ sat\ \phi$ the derivability of the correctness assertion $A\ sat\ \phi$ in the proof system introduced in the previous section (assuming as additional axioms in rule **T7** all valid temporal formulas). The following theorem states the soundness and (relative) completeness of this proof system.

Theorem 1. *We have* $\vdash_p A\ sat\ \phi$ iff $\models A\ sat\ \phi$, *for every correctness assertion* $A\ sat\ \phi$.

At the heart of this theorem lies the compositionality of the semantics R' which follows from the compositionality of the underlying semantics R as described in [5]. Given the compositionality of the semantics R' soundness can be proved by induction on the length of the derivation. As for completeness, following the standard notion for Hoare-style proof systems as introduced by [12] we consider here a notion of relative completeness. We assume the existence of a property which describes exactly the denotation of a process, that is, we assume that for any process A there exists a formula $\psi(A)$, such that $\rho \in R'(A)$ iff, for any γ, $\rho \models_\gamma \psi(A)$ holds[4]. This is analogous to assume the expressibility of the strongest postcondition of a process P, as with standard Hoare-like proof systems. Furthermore, we assume as additional axioms all the valid temporal formulas, (for use in the consequence rule). Also this assumption, in general, is needed to obtain completeness of Hoare logics. Using these assumptions, the proof of completeness follows the lines of the analogous proof in [4].

5 Related Work

A simpler temporal logic for tccp has been defined in [7] by considering epistemic operators of "belief" and "knowledge" which corresponds to the operators I and O considered in the present paper. Even though the intuitive ideas of the two papers are similar, the technical treatment is different. In fact, the logic in [7] is

[4] In order to describe recursion, the syntax of the temporal formulas has to be extended with a fixpoint operator of the form $\mu p(x).\phi$, where $p(x)$ is supposed to occur positively in ϕ and the variable x denotes the formal parameter associated with the procedure p (see [4]). The meaning of $\mu p(x).\phi$ is given by a least fixpoint-construction which is defined in terms of the lattice of sets of sequences of predicate assignements ordered by set-inclusion.

less expressive than the present one, since it does not allow constraint (predicate) variables. As a consequence, the proof system defined in [7] was not complete.

Recently, a logic for a different timed extension of ccp, called ntcc, has been presented in [23]. The language ntcc [29,21] is a non deterministic extension of the timed ccp language defined in [26]. Its computational model, and therefore the underlying logic, are rather different from those that we considered. Analogously to the case of the ESTEREL language, computation in ntcc (and in the language defined in [26]) proceeds in "bursts of activity": in each phase a ccp process is executed to produce a response to an input provided by the environment. The process accumulates monotonically information in the store, according to the standard ccp computational model, until it reaches a "resting point", i.e. a terminal state in which no more information can be generated. When the resting point is reached, the absence of events can be checked and it can trigger actions in the next time interval. Thus, each time interval is identified with the time needed for a ccp process to terminate a computation. Clearly, in order to ensure that the next time instant is reached, the ccp program has to be always terminating, thus it is assumed that it does not contain recursion (a restricted form of recursion is allowed only across time boundaries). Furthermore, 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, unless they are argument to a specific primitive. These assumptions allow to obtain an elegant semantic model consisting of sequences of sets of resting points (each set describing the behavior 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 allows us to obtain a direct timed extension of ccp which maintain the essential features of ccp computations. No restriction on recursion is needed to ensure that the next time instant is reached, since at each time instant there are only a finite number of parallel agents which can perform a finite number of (ask and tell) actions. Also, no explicit transfer of information across time boundaries is needed in *tccp*, since the (monotonic) evolution of the store is the same as in ccp (these differences affects the expressive power of the language, see [5] 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 [5]).

From a logical point of view, as shown in [4] 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 [23] 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. A proof system for proving the resulting linear temporal properties is also defined in [23]. 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 arises when one wants to describe the input/output behavior 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 behavior corresponding to the interaction of the environment, which provides the input, with the process, which produces the output. This is reflected in the the logic we have defined.

Related to the present paper is also [13], 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 behavior of the tcc program, thus allowing the application of existing model checking algorithms.

6 Conclusions

We introduced a temporal logic for reasoning about the correctness of a timed extension of ccp and we proved the soundness and (relative) completeness of a related proof system. As discussed in the previous section, due to the need to characterize the input/output behaviour of processes (rather than the resting points) our logic is rather complex. Therefore it is important to investigate possible axiomatizations and decision procedures for this logic (for example considering a semantic tableaux method). We are currently investigating these issues, also in order to assess the practical usability of our proof-system (as the consequence rule requires a certain implication in the logic to be valid). Since *reactive sequences* have been used also in the semantics of several other languages, including dataflow and imperative ones [20,9,8,11,6], we plan also to consider extensions of our logic to deal with these different languages.

References

1. L. Aceto and D. Murphy. Timing and causality in process algebra. *Acta Informatica*, 33(4): 317-350, 1996.
2. J. Baeten and J. Bergstra. Real time process algebra. *Formal Aspects of Computing*, 3(2): 142-188, 1991.
3. G. Berry and G. Gonthier. The ESTEREL programming language: Design, semantics and implementation. *Science of Computer Programming*, 19(2):87-152, 1992.
4. F.S. de Boer, M. Gabbrielli, E. Marchiori and C. Palamidessi. Proving Concurrent Constraint Programs Correct. *TOPLAS*, 19(5): 685-725. ACM Press, 1997.
5. F.S. de Boer, M. Gabbrielli and M.C. Meo. A Timed CCP Language. *Information and Computation*, 161, 2000.
6. F. S. de Boer, M. Gabbrielli, and M. C. Meo. A Denotational Semantics for a Timed Linda Language. In *Proc. PPDP 2001*. ACM Press, 2001.

7. F.S. de Boer, M. Gabbrielli and M.C. Meo. A Temporal Logic for reasoning about Timed Concurrent Constraint Programs. In *Proc. TIME 01.* IEEE Press, 2001.
8. 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 *Proc. of CONCUR'91*, vol. 527 of *LNCS*, pages 111–126. Springer-Verlag, 1991.
9. 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.
10. P. Bremond-Gregoire and I. Lee. A Process Algebra of Communicating Shared Resources with Dense Time and Priorities. *Theoretical Computer Science* 189, 1997. Springer-Verlag, 1997.
11. S. Brookes. A fully abstract semantics of a shared variable parallel language. In *Proc. Eighth LICS.* IEEE Computer Society Press, 1993.
12. S. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal of Computation 7,* 1, 70–90.
13. M. Falaschi, A. Policriti, A. Villanueva. Modeling Concurrent systems specified in a Temporal Concurrent Constraint language. in *Proc. AGP'2000.* 2000.
14. M. Fisher. An introduction to Executable Temporal Logics. *Knowledge Engineering Review*, 6(1): 43-56, 1996.
15. N. Halbwachs, P. Caspi, and D. Pilaud. The synchronous programming language LUSTRE. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, 1991.
16. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, pages 231-274, 1987.
17. L. Henkin, J.D. Monk, and A. Tarski. *Cylindric Algebras (Part I).* North-Holland, 1971.
18. M. Hennessy and T. Regan. A temporal process algebra. *Information and Computation*, 117: 221-239, 1995.
19. P. Le Guernic, M. Le Borgue, T. Gauthier, and C. Le Marie. Programming real time applications with SIGNAL. In *Special issue on Another Look at Real-time Systems*, Proceedings of the IEEE, 1991.
20. 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.
21. M. Nielsen and F.D. Valencia. The ntcc Calculus and its applications. Draft, 2001.
22. Z. Manna and A. Pnueli. *The temporal logic of reactive systems.* Springer-Verlag, 1991.
23. C. Palamidessi and F.D. Valencia. A Temporal Concurrent Constraint Programming Calculus. In *Proc. CP 01*, LNCS 2239, pag. 302-316. Springer-Verlag, 2001.
24. V.A. Saraswat. *Concurrent Constraint Programming Languages.* PhD thesis, Carnegie-Mellon University, January 1989. Published by The MIT Press, 1991.
25. V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of POPL*, pages 232–245. ACM Press, 1990.
26. V.A. Saraswat, R. Jagadeesan, and V. Gupta Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation*, 22(5-6):475–520, 1996.
27. V.A. Saraswat, M. Rinard, and P. Panangaden. Semantics foundations of Concurrent Constraint Programming. In *Proc. of POPL.* ACM Press, 1991.
28. G. Smolka. The Definition of Kernel Oz. In A. Podelski editor, *Constraints: Basics and Trends*, vol. 910 of *LNCS*, pages 251–292. Springer-Verlag, 1995.
29. F.D. Valencia. Reactive Constraint Programming. *Brics Progress Report*, June 2000.