A Temporal Concurrent Constraint Programming Calculus

Catuscia Palamidessi* and Frank D. Valencia

Penn State University, USA. Email:catuscia@cse.psu.edu
BRICS**, University of Aarhus, Denmark. Email:fvalenci@brics.dk

Abstract The tcc model is a formalism for reactive concurrent constraint programming. In this paper we propose a model of *temporal concurrent constraint programming* which adds to tcc the capability of modeling asynchronous and non-deterministic timed behavior. We call this tcc extension the *ntcc calculus*. The expressiveness of ntcc is illustrated by modeling cells, asynchronous bounded broadcasting and timed systems such as RCX controllers. We present a denotational semantics for the strongest-postcondition of ntcc processes and, based on this semantics, we develop a proof system for linear temporal properties of these processes.

1 Introduction

The tcc model [16] is a formalism for reactive ccp which combines deterministic ccp [18] with ideas from the Synchronous Languages [2]. Time is conceptually divided into *discrete intervals (or time-units)*. In a particular time interval, a deterministic ccp process receives a stimulus (i.e. a constraint) from the environment, it executes with this stimulus as the initial store, and when it reaches its resting point, it responds to the environment with the resulting store. Also the resting point determines a residual process, which is then executed in the next time interval.

The tcc model is inherently deterministic and synchronous. Indeed, patterns of temporal behavior such as "the system must output c *within* the next t time units" or "the message must be delivered but *there is no bound* in the delivery time" cannot be expressed within the model. It also rules out the possibility of choosing one among several alternatives as an output to the environment. The task of *zigzagging* (see Section 4), in which a robot can unpredictably choose its next move, is an example where non-determinism is useful.

In general, a benefit of allowing the specification of non-deterministic behavior is to free programmers from the necessity of coping with issues that are irrelevant to the problem specification. Dijkstra's language of guarded commands, for

* The contribution of Catuscia Palamidessi to this research has been supported by the NSF-POWRE grant EIA-0074909.
** Basic Research in Computer Science, Centre of the Danish National Research Foundation.

example, uses a nondeterministic construction to help free the programmer from over-specifying a method of solution. As pointed out in [21], a disciplined use of nondeterminism can lead to a more straightforward presentation of programs. This view is consistent with the declarative flavor of ccp: The programmer specifies by means of constraints the possible values that the program variables can take, without being required to provide a computational procedure to enforce the corresponding assignments.

Futhermore, a very important benefit of allowing the specification of non-deterministic and asynchronous behavior arises when modeling the interaction among several components running in parallel, in which one component is part of the environment of the others. These systems often need non-determinism and asynchrony to be modeled faithfully.

In this paper we propose an extension of tcc, which we call the *ntcc calculus*, for temporal ccp. The calculus is obtained by adding *guarded-choice* for modeling non-determinism and an *unbounded but finite delay* operator for asynchrony. Computation in ntcc progresses as in tcc, except for the non-determinism and asynchrony induced by the new constructs. The calculus allows for the specification of temporal properties, and for modeling and expressing constraints upon the environment both of which are useful in proving properties of timed systems. We shall illustrate the expressiveness of ntcc by modeling constructs such as cells, asynchronous bounded broadcasting and some applications involving RCX$^{\text{TM}}$ controllers.

The declarative nature of ntcc comes to the surface when we consider the denotational characterization of the strongest postcondition of a process, as defined in [5] for ccp, and extend it to a timed setting. We show that the elegant model based on closure operators, developed in [18] for deterministic ccp, can be extended to a simple sound model for ntcc. We also obtain completeness for a fragment we shall call local-independent choice.

The logical nature of ntcc comes to the surface when we consider its relation with linear temporal logic: All the operators of ntcc correspond to temporal logic constructs. We develop a sound system for linear temporal properties of ntcc and show that the system is also (relatively) complete wrt local-independent choice processes. Our system is then complete for tcc as well, since every tcc process falls into the category of local-independent choice ntcc processes.

The main contributions of this paper can be summarized as follows: (1) a model of temporal ccp more expressive than tcc (2) a denotational semantics for the strongest postcondition of ntcc processes, and (3) a proof system for linear temporal properties of ntcc process.

2 The Calculus

In this section we present the syntax and an operational semantics of the ntcc calculus. First we recall the notion of constraint system.

Basically, a constraint system provides a signature from which syntactically denotable objects in language called *constraints* can be constructed, and an entailment relation specifying interdependencies between such constraints.

Definition 1 (Constraint Systems). *A constraint system is a pair (Σ, Δ) where Σ is a signature specifying functions and predicate symbols, and Δ is a consistent first order theory.*

Given a constraint system (Σ, Δ), let $\mathcal{L}$ be the underlying first-order language $(\Sigma, \mathcal{V}, \mathcal{S})$, where $\mathcal{V} = \{x, y, z, \dots\}$ is a countable set of variables and $\mathcal{S}$ is the set containing the symbols $\dot{\neg}, \dot{\wedge}, \dot{\Rightarrow}, \dot{\exists}, \texttt{true}$ and $\texttt{false}$ which denote logical negation, conjunction, implication, existential quantification, and the always true and always false predicates, respectively. *Constraints,* denoted by $c, d, \dots$ are first-order formulae over $\mathcal{L}$. We say that c *entails* d in Δ, written $c \vdash_\Delta d$ (or just $c \vdash d$ when no confusion arises), if $c \Rightarrow d$ is true in all models of Δ. We write $c \approx d$ iff $c \vdash d$ and $d \vdash c$. We will consider constraints modulo $\approx$ and use $\mathcal{C}$ for the set of representants of equivalence classes of constraints. For operational reasons we shall require $\vdash$ to be decidable.

Process Syntax. Processes $P, Q, \ldots \in \mathit{Proc}$ are built from constraints $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system by the following syntax.

$$\begin{array}{rl} P, Q, \ldots ::= & \mathbf{tell}(c) \ \mid \sum\limits_{i \in I} \mathbf{when}\, c_i \,\mathbf{do}\, P_i \mid P \parallel Q \ \mid \mathbf{local}\ x\ \mathbf{in}\, P \\ \mid & \mathbf{next}\, P \mid \mathbf{unless}\ c\ \mathbf{next}\, P \mid !\, P \quad \mid \star P \ . \end{array}$$

The only move or action of process $\mathbf{tell}(c)$ is to add the constraint c to the current store, thus making c available to other processes in the current time interval. The guarded-choice $\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$, where I is a finite set of indexes, represents a process that, in the current time interval, must non-deterministically choose one of the P_j $(j \in I)$ whose corresponding constraint c_j is entailed by the store. The chosen alternative, if any, precludes the others. If no choice is possible then the summation is precluded. We use $\sum_{i \in I}\ P_i$ as an abbreviation for the "blind-choice" process $\sum_{i \in I} \mathbf{when}\,(\texttt{true})\,\mathbf{do}\, P_i$. We use **skip** as an abbreviation of the empty summation and "+" for binary summations.

Process $P \parallel Q$ represents the parallel composition of P and Q. In one time unit (or interval) P and Q operate concurrently, "communicating" via the common store. We use $\prod_{i \in I} P_i$, where I is finite, to denote the parallel composition of all P_i. Process **local** x **in** P behaves like P, except that all the information on x produced by P can only be seen by P and the information on x produced by other processes cannot be seen by P.

The process $\mathbf{next}\, P$ represents the activation of P in the next time interval. Hence, a move of $\mathbf{next}\, P$ is a unit-delay of P. The process $\mathbf{unless}\, c\, \mathbf{next}\, P$ is similar, but P will be activated only if c cannot be inferred from the current store. The "unless" processes add (weak) time-outs to the calculus, i.e., they wait one time unit for a piece of information c to be present and if it is not, they trigger activity in the next time interval. We use $\mathbf{next}^n(P)$ as an abbreviation for $\mathbf{next}(\mathbf{next}(\ldots(\mathbf{next}\,P)\ldots))$, where **next** is repeated n times.

The operator "!" is a delayed version of the replication operator for the $\pi-$calculus ([14]): $!\,P$ represents $P \parallel \mathbf{next}\,P \parallel \mathbf{next}^2 P \parallel \ldots$, i.e. unboundely many copies of P but one at a time. The replication operator is the only way of defining infinite behavior through the time intervals.

The operator "$\star$" corresponds to the unbounded but finite delay operator ϵ for synchronous CCS ([13]) and it allows us to express asynchronous behavior through the time intervals. The process $\star\, P$ represents an arbitrary long but finite delay for the activation of P. For example, $\star\,\mathbf{tell}(c)$ can be viewed as a message c that is eventually delivered but there is no upper bound on the delivery time. By using the $\star$ operator we can define a *fair asynchronous* parallel composition $P \mid Q$ as $(P \parallel \star\, Q)\ +\ (\star\, P \parallel Q)$ as described in [13]. A move of $P \mid Q$ is either one of P or one of Q (or both). Moreover, both P and Q are eventually executed (i.e. a fair execution of $P \mid Q$).

We shall use $!_I P$ and $\star_I P$, where I is an interval of the natural numbers, as an abbreviation for $\prod_{i\in I} \mathbf{next}^i P$ and $\sum_{i \in I} \mathbf{next}^i P$, respectively. For instance, $\star_{[m,n]} P$ means that P is eventually active between the next m and $m+n$ time units, while $!_{[m,n]} P$ means that P is always active between the next m and $m+n$ time units.

Operational Semantics. Operationally, the current information is represented as a constraint $c \in \mathcal{C}$, so-called *store*. Our operational semantics is given by considering transitions between *configurations* γ of the form $\langle P, c\rangle$. We define Γ as the set of all configurations. Following standard lines, we extend the syntax with a construct $\mathbf{local}\,(x,d)\,\mathbf{in}\,P$, which represents the evolution of a process of the form $\mathbf{local}\,x\,\mathbf{in}\,Q$, where d is the local information (or store) produced during this evolution. Initially d is "empty", so we regard $\mathbf{local}\,x\,\mathbf{in}\,P$ as $\mathbf{local}\ (x, \mathtt{true})\ \mathbf{in}\ P$

We need to introduce a notion of free variables that is invariant wrt the equivalence on constraints. We can do so by defining the "relevant" free variables of c as $fv(c) = \{x \in \mathcal{V} \mid \exists_x c \not\approx c\}$. For the bound variables, define $bv(c) = \{x \in \mathcal{V} \mid x \text{ occurs in } c\} - fv(c)$. Regarding processes, define $fv(\mathbf{tell}(c)) = fv(c)$, $fv(\sum_i \mathbf{when}\ c_i\ \mathbf{do}\ P_i) = \bigcup_i fv(c_i) \cup fv(P_i)$, $fv(\mathbf{local}\,x\,\mathbf{in}\,P) = fv(P) - \{x\}$. The bound variables and the other cases are defined analogously.

Definition 2 (Structural Congruence). *Let $\equiv$ be the smallest congruence over processes satisfying the following laws:*

1. $(Proc/_{\equiv}, \parallel, \mathbf{skip})$ *is a symmetric monoid.*
2. $P \equiv Q$ *if they only differ by a renaming of bound variables.*
3. $\mathbf{next}\,\mathbf{skip}\ \equiv\ \mathbf{skip}\qquad \mathbf{next}(P \parallel Q) \equiv \mathbf{next}\,P \parallel \mathbf{next}\,Q.$
4. $\mathbf{local}\,x\,\mathbf{in}\,\mathbf{skip} \equiv \mathbf{skip}\quad \mathbf{local}\,x\,y\,\mathbf{in}\,P \equiv \mathbf{local}\,y\,x\,\mathbf{in}\,P.$
5. $\mathbf{local}\,x\,\mathbf{in}\,\mathbf{next}\,P\ \equiv \mathbf{next}(\mathbf{local}\,x\,\mathbf{in}\,P).$
6. $\mathbf{local}\,x\,\mathbf{in}\,(P \parallel Q) \equiv\ P\ \parallel\ \mathbf{local}\,x\,\mathbf{in}\,Q\quad$ *if* $\quad x \notin fv(P).$

We extend $\equiv$ to configurations by defining $\langle P, c\rangle \equiv \langle Q, c\rangle$ if $P \equiv Q$.

The reduction relations $\longrightarrow \subseteq \Gamma \times \Gamma$ and $\Longrightarrow \subseteq Proc \times \mathcal{C} \times \mathcal{C} \times Proc$ are the least relations satisfying the rules appearing in Table 1. The *internal transition* $\langle P, c\rangle \longrightarrow \langle Q, d\rangle$ should be read as "P with store c reduces, in one internal step, to Q with store d". The *observable transition* $P \xRightarrow{(c,d)} Q$ should be read as "P on input c reduces, in one time unit, to Q with store d". As in tcc, the store does not transfer automatically from one interval to another.

We now give a description of the operational rules. Rules TELL, CHOICE, PAR and LOC are standard [18]. Rule UNLESS says that if c is entailed by the current store, then the execution of the process P (in the next time interval) is precluded. Rule REPL specifies that the process $!\,P$ produces a copy P at the current time unit, and then persists in the next time unit. STAR says that $\star P$ triggers P in some time interval (either in the current one or in a future one). Rule STRUCT simply says that structurally congruent processes have the same reductions.

Rule OBS says that an observable transition from P labeled by (c, d) is obtained by performing a terminating sequence of internal transitions from $\langle P, c\rangle$ to $\langle Q, d\rangle$, for some Q. The process to be executed in the next time interval, $F(Q)$ ("future" of Q), is obtained by removing from Q what was meant to be executed only in the current time interval and any local information which has been stored in Q, and by "unfolding" the sub-terms within **next** R expressions. More precisely:

Definition 3 (Future Function). *The partial function $F : Proc \rightharpoonup Proc$ is defined as follows:*

$$F(P) = \begin{cases} Q & \textit{if } P = \mathbf{next}\ Q \textit{ or } P = \mathbf{unless}\ c\ \mathbf{next}\ Q \\ F(P_1) \parallel F(P_2) & \textit{if } P = P_1 \parallel P_2 \\ \mathbf{local}\ x\ \mathbf{in}\ F(Q) & \textit{if } P = \mathbf{local}\,(x, c)\ \mathbf{in}\ Q \\ \mathbf{skip} & \textit{if } P = \sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i . \end{cases}$$

Remark: Function F does not need to be total since whenever we apply F to a process P (Rule OBS in Table 1), all replications and unbounded finite-delay operators in P occur within a next construction.

Interpreting Process Runs. Let us consider an infinite sequence of observable transitions

$$P \;=\; P_1 \xRightarrow{(c_1,c_1')} P_2 \xRightarrow{(c_2,c_2')} P_3 \xRightarrow{(c_3,c_3')} \ldots$$

This sequence can be interpreted as a *interaction* between the system P and an environment. At the time unit i, the environment provides a *stimulus* c_i and P_i produces c_i' as *response*. If $\alpha = c_1.c_2.c_3\ldots.$ and $\alpha' = c_1'.c_2'.c_3'\ldots$, we represent the above interaction as $P \xRightarrow{(\alpha,\alpha')}{}^{\omega}$.

Alternatively, if $\alpha = \texttt{true}^{\omega}$, we can interpret the run as an interaction among the parallel components in P without the influence of an external environment

TELL	$\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d \dot\wedge c\rangle$
CHOICE	$\left\langle \sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i, d\right\rangle \longrightarrow \langle P_j, d\rangle \quad \text{if } d \vdash c_j, \text{ for } j \in I$
PAR	$\dfrac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle}$
LOC	$\dfrac{\left\langle P, c \dot\wedge \dot\exists_x d\right\rangle \longrightarrow \langle Q, c'\rangle}{\langle \mathbf{local}\ (x, c)\ \mathbf{in}\ P, d\rangle \longrightarrow \left\langle \mathbf{local}\ (x, c')\ \mathbf{in}\ Q, d \dot\wedge \dot\exists_x c'\right\rangle}$
UNLESS	$\langle \mathbf{unless}\ c\ \mathbf{next}\ P, d\rangle \longrightarrow \langle \mathbf{skip}, d\rangle \quad \text{if } d \vdash c$
REPL	$\langle !\, P, c\rangle \longrightarrow \langle P \parallel \mathbf{next}\, !\, P, c\rangle$
STAR	$\langle \star\, P, c\rangle \longrightarrow \langle \mathbf{next}^n P, c\rangle \quad \text{for some } n \geq 0.$
STRUCT	$\dfrac{\gamma_1 \equiv \gamma_1' \quad \gamma_1' \longrightarrow \gamma_2' \quad \gamma_2' \equiv \gamma_2}{\gamma_1 \longrightarrow \gamma_2}$
OBS	$\dfrac{\langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow}{P \xRightarrow{(c,d)} F(Q)}$

Table1. An operational semantics for ntcc. The upper part defines the internal transitions while the lower part defines the observable transitions. The function F, used in OBS, is given in Definition 3

(i.e., each component is part of the environment of the others). In this case α is called the *empty* input sequence and α' is regarded as a *timed* observation of such an interaction in P.

3 Strongest Postcondition: Denotation and Logic

In this section we introduce the *strongest postcondition* of a process and investigate its denotational and logic. Henceforward, we use α, α' to represent elements of $\mathcal{C}^\omega$ and β to represent an element of $\mathcal{C}^*$. Given $c \in \mathcal{C}$, $c.\alpha$ represents the concatenation of c and α. Furthermore, $\beta.\alpha$ represents the concatenation of β and α. We use $\dot\exists_x \alpha$ to represent the sequence obtained by applying $\dot\exists_x$ to each constraint in α. Notation $\alpha(i)$ denotes the i-th element in α.

We define the *strongest postcondition* of P, $sp(P)$, as the set of all sequences P can possibly output. More precisely,

D1	$[\![\mathbf{tell}(c)]\!] \;=\; \{d.\alpha \mid d \vdash c, \alpha \in \mathcal{C}^\omega\}$
D2	$[\![\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i]\!] \;=\; \bigcup_{i\in I}\{d.\alpha \mid d \vdash c_i, d.\alpha \in [\![P_i]\!]\} \cup \bigcap_{i\in I}\{d.\alpha \mid d \not\vdash c_i, d.\alpha \in \mathcal{C}^\omega\}$
D3	$[\![P \parallel Q]\!] \;=\; [\![P]\!] \cap [\![Q]\!]$
D4	$[\![\mathbf{local}\ x\ \mathbf{in}\ P]\!] \;=\; \{\alpha \mid \text{there exists } \alpha' \in [\![P]\!] \text{ s.t. } \dot{\exists}_x\alpha = \dot{\exists}_x\alpha'\}$
D5	$[\![\mathbf{next}\ P]\!] \;=\; \{d.\alpha \mid d \in \mathcal{C}, \alpha \in [\![P]\!]\}$
D6	$[\![\mathbf{unless}\ c\ \mathbf{next}\ P]\!] \;=\; \{d.\alpha \mid d \vdash c, \alpha \in \mathcal{C}^\omega\} \cup \{d.\alpha \mid d \not\vdash c, \alpha \in [\![P]\!]\}$
D7	$[\![!\,P]\!] \;=\; \{\alpha \mid \forall\beta \in \mathcal{C}^*, \alpha' \in \mathcal{C}^\omega \text{ s.t. } \alpha = \beta.\alpha', \text{ we have } \alpha' \in [\![P]\!]\}$
D8	$[\![\star\, P]\!] \;=\; \{\beta.\alpha \mid \beta \in \mathcal{C}^*, \alpha \in [\![P]\!]\}$

Table2. Denotational Semantics of ntcc

Definition 4 (Strongest Postcondition). *Let us define the set $sp(P)$ as $\{\alpha' \mid P \xrightarrow{(\alpha,\alpha')}{}^{\omega}$ for some $\alpha\}$.*

Denotational Semantics. We give now a denotational characterization of the strongest postcondition following ideas in [5] and [16] for the ccp and tcc case, respectively. The presence of non-determinism, however, presents a technical problem to deal with: The strongest postcondition for the hiding operator cannot be specified compositionally (see [5]). Therefore, we will have to identify a practical fragment for which the semantics is complete.

The denotational semantics is defined as a function $[\![\cdot]\!]$ which associates to each process a set of infinite constraint sequences, namely $[\![\cdot]\!] : Proc \to \mathcal{P}(\mathcal{C}^\omega)$. The definition of this function is given in Table 2. Intuitively, $[\![P]\!]$ is meant to capture the set of all sequences P can possibly output. For instance, the sequences that $\mathbf{tell}(c)$ can output are those whose first element is stronger than c (D1). Process $\mathbf{next}\,P$ has not influence in the first element of a sequence, thus $d.\alpha$ can be output by it if α is can be output by P (D5). A sequence can be output by $!\,P$ if every suffix of it can be output by P (D7). The other rules can be explained analogously. The next theorems state the relation between the denotation of P and its strongest postcondition.

Theorem 1 (Soundness). *For every ntcc process P, $sp(P) \subseteq [\![P]\!]$.*

For the reasons mentioned at the beginning of this section, the converse of this theorem does not hold in general. Nevertheless, it holds for *local-independent choice* processes which we define next.

Definition 5 (Local-Independent Choice). *A process P is said to be* local-independent choice *iff for all* **local** x **in** Q *in P, for all* $\sum_{i\in I}$ **when** c_i **do** Q_i *in Q, the c_i's either are equivalent, mutually exclusive or do not have free occurrences of x.*

This is a substantial fragment of ntcc since every restricted-choice process is also local-independent choice and, unlike the restricted-choice fragment defined [7], its condition does not imply structural confluence. In fact, all the process examples in this paper are local-independent choice.

Theorem 2 (Completeness). *If P is a local-independent choice ntcc process, then $sp(P) = [\![P]\!]$.*

For deterministic processes such as tcc processes, namely those which contain neither the choice (except when the index set is a singleton) nor the $\star$ operator, we have an even stronger result: the semantics allows to retrieve the input-output relation (which for deterministic processes is a function). Let us use $\leq$ to denote the (partial) order relation $\{(\alpha, \alpha') \mid \forall i \geq 1 \;\; \alpha'(i) \vdash \alpha(i)\}$ and $min(S)$ to denote the minimal element of $S \subseteq \mathcal{C}^\omega$ in the complete lattice $(\mathcal{C}^\omega, \leq)$.

Theorem 3. *If P is a deterministic process, then $(\alpha, \alpha') \in io(P)$ iff $\alpha' = min([\![P]\!] \cap \uparrow \alpha)$, where $\uparrow \alpha = \{\alpha'' | \alpha \leq \alpha''\}$.*

Linear-Temporal Logic. Let us define a linear temporal logic for expressing properties of ntcc processes. The formulae $A, B, ... \in \mathcal{A}$ are defined by the grammar $A ::= c \mid A \Rightarrow A \mid \neg A \mid \exists_x A \mid \circ A \mid \Box A \mid \Diamond A$. The symbol c denotes an arbitrary constraint. The symbols $\Rightarrow$, $\neg$ and $\exists_x$ represent temporal logic implication, negation and existential quantification. These symbols are not to be confused with the logic symbols $\dot{\Rightarrow}$, $\dot{\neg}$ and $\dot{\exists}_x$ of the constraint system. The symbols $\circ$, $\Box$, and $\Diamond$ denote the temporal operators *next*, *always* and *sometime*. We use $A \vee B$ as an abbreviation of $\neg A \Rightarrow B$ and $A \wedge B$ as an abbreviation of $\neg(\neg A \vee \neg B)$.

The standard interpretation structures of linear temporal logic are infinite sequences of states [12]. In ntcc states are represented with constraints, thus we consider as interpretations the elements of $\mathcal{C}^\omega$. We say that $\alpha \in \mathcal{C}^\omega$ is a model of A, notation $\alpha \models A$, if $\langle \alpha, 1 \rangle \models A$, where:

$$
\begin{array}{lll}
\langle \alpha, i \rangle \models c & \text{iff} & \alpha(i) \vdash c \\
\langle \alpha, i \rangle \models \neg A & \text{iff} & \langle \alpha, i \rangle \not\models A \\
\langle \alpha, i \rangle \models A_1 \Rightarrow A_2 & \text{iff} & \langle \alpha, i \rangle \models A_1 \text{ implies } \langle \alpha, i \rangle \models A_2 \\
\langle \alpha, i \rangle \models \circ A & \text{iff} & \langle \alpha, i+1 \rangle \models A \\
\langle \alpha, i \rangle \models \Box A & \text{iff} & \text{for all } j \geq i \;\; \langle \alpha, j \rangle \models A \\
\langle \alpha, i \rangle \models \Diamond A & \text{iff} & \text{there exists } j \geq i \;\text{ s.t. } \langle \alpha, j \rangle \models A \\
\langle \alpha, i \rangle \models \exists_x A & \text{iff} & \text{there exists } \alpha' \in \mathcal{C}^\omega \text{ s.t. } \dot{\exists}_x \alpha = \dot{\exists}_x \alpha' \text{ and } \langle \alpha', i \rangle \models A.
\end{array}
$$

We define $[\![A]\!] = \{\alpha \mid \alpha \models A\}$, i.e., the collection of all models of A.

P1	$\mathbf{tell}(c) \ \vdash\ c$	P3	$\dfrac{P \ \vdash\ A \qquad Q \ \vdash\ B}{P \parallel Q \ \vdash\ A \wedge B}$
P2	$\dfrac{\forall i \in I \quad P_i \ \vdash\ A_i}{\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i \ \vdash\ \bigvee_{i \in I}(c_i \wedge A_i) \vee \bigwedge_{i \in I} \neg c_i}$	P4	$\dfrac{P \ \vdash\ A}{\mathbf{local}\ x\ \mathbf{in}\ P \ \vdash\ \exists_x A}$
P5	$\dfrac{P \ \vdash\ A}{\mathbf{next}\ P \ \vdash\ \circ A}$	P6	$\dfrac{P \ \vdash\ A}{\mathbf{unless}\ c\ \mathbf{next}\ P \ \vdash\ c \vee \circ A}$
P7	$\dfrac{P \ \vdash\ A}{!\,P \ \vdash\ \Box A}$	P8	$\dfrac{P \ \vdash\ A}{\star P \ \vdash\ \Diamond A}$
P9	$\dfrac{P \ \vdash\ A}{P \ \vdash\ B}$ if $A \Rightarrow B$		

Table3. A proof system for linear temporal properties of ntcc processes

Proving Properties of Processes. We are interested in assertions of the form $P \vdash A$, whose intuitive meaning is that every sequence P can possibly output satisfies the property expressed by A – i.e., that every sequence in $sp(P)$ (Definition 4) is a model of A. An inference system for such assertions is presented in Table 3. We will say that $P \ \vdash\ A$ holds if the assertion $P \ \vdash\ A$ has a proof in this system.

The following theorem states the soundness and the relative completeness of the proof system.

Theorem 4 (Relative Completeness). *For every ntcc process P and every formula A, $P \ \vdash\ A$ holds iff $[\![P]\!] \subseteq [\![A]\!]$ holds.*

The reason why this theorem is called "relative completeness" is because of the consequence rule P9 (consequence rule). Proving $A \Rightarrow B$ is known to be decidable for the quantifier-free fragment of linear time temporal formulae as well as for some other interesting first-order fragments (see [10]).

>From Theorems 4, 1 and 2 we immediately derive the following:

Corollary 1. *1. For every ntcc process P and every formula A, if $P \ \vdash\ A$ holds then $sp(P) \subseteq [\![A]\!]$ holds.*

2. For every local-independent choice ntcc process P and every formula A, $P \vdash A$ holds iff $sp(P) \subseteq [\![A]\!]$ holds.

We shall see that the kind of recursion considered in [16] can be encoded in ntcc. Hence, tcc processes can be considered as a particular case of local-independent choice ntcc processes, and therefore the proof system is complete for tcc.

The following notion will be useful in the Section 4, for discussing properties of our examples.

Definition 6 (Strongest Derivable Formulae). *A formula A is the strongest temporal formula derivable for P if $P \vdash A$ and for all A' such that $P \vdash A'$, we have $A \Rightarrow A'$.*

Note that the strongest temporal formula of a process P is unique modulo logical equivalence. We give now a constructive definition of such formula.

Definition 7 (Strongest Temporal Formula Function). *Let the function $\mathit{stf} : \mathit{Proc} \to \mathcal{A}$ be defined as follows:*

$$
\begin{array}{lcl}
\mathit{stf}(\mathbf{tell}(c)) & = & c \\
\mathit{stf}(\sum_{i\in I} \mathbf{when}\,(c_i)\,\mathbf{do}\,P_i) & = & \big(\bigvee_{i\in I} c_i \wedge \mathit{stf}(P_i)\big) \vee \bigwedge_{i\in I} \neg c_i \\
\mathit{stf}(P \parallel Q) & = & \mathit{stf}(P) \wedge \mathit{stf}(Q) \\
\mathit{stf}(\mathbf{local}\, x\, P) & = & \exists_x \mathit{stf}(P) \\
\mathit{stf}(\mathbf{next}\ P) & = & \circ\, \mathit{stf}(P) \\
\mathit{stf}(\mathbf{unless}\ c\ \mathbf{next}\ P) & = & c \vee \circ\, \mathit{stf}(P) \\
\mathit{stf}(!\,P) & = & \Box\, \mathit{stf}(P) \\
\mathit{stf}(\star\, P) & = & \Diamond\, \mathit{stf}(P).
\end{array}
$$

We can easily prove that $[\![\mathit{stf}(P)]\!] = [\![P]\!]$ and that $P \vdash \mathit{stf}(P)$. From these we have:

Proposition 1. *For every process P, $\mathit{stf}(P)$ is the strongest temporal formula derivable for P.*

Note that to prove that $P \vdash A$ is sufficient to prove that $\mathit{stf}(P) \Rightarrow A$. However, to prove such implication may not be always feasible or possible. The proof system provides the additional flexibility of proving $P \vdash A$ by using the consequence rule (P9) on subprocesses of P and on formulae different from A.

4 Applications

In this section we illustrate some ntcc examples. We first need to define an underlying constraint system.

Definition 8 (A Finite-Domain Constraint System). *Let max be a positive integer number. Define $FD[max]$ as the constraint system whose signature Σ includes symbols in $\{0, \mathrm{succ}, +, \times, =\}$ and the first-order theory Δ is the set of sentences valid in arithmetic modulo max.*

The intended meaning of $FD[max]$ is the natural numbers interpreted as in arithmetic modulo max. Henceforth, we assume that the signature is extended with two new unary predicate symbols call and change. We will designate Dom as the set $\{0, 1, ..., max-1\}$ and use v and w to range over its elements.

Recursion. We can encode recursive definitions of the form $q(x) \stackrel{\text{def}}{=} P_q$, where q is the process name and P_q contains at most one occurrence of q which must be within the scope of a "**next**" and out of the scope of any "!". The reason for such a restriction is that we want to keep bounded the response time of the system.

We also want to consider the call-by-value. This may look unnatural since in constraint programming the natural parameter passing mechanism is through "logical variables", like in logic programming. Indeed, it is more difficult to encode in ntcc call-by-value than "call-by-logical-variable". However, for the kind of applications we have in mind (some of which are illustrated in the rest of this section), call-by-value is the mechanism we need. Note also that we mean call-by-value in the sense of value "persisting through the time intervals", and this would not be possible to achieve directly with the "call-by-logical-variable", because the values of variables are not maintained from one interval to the next. More precisely: The intended behavior of a call $q(t)$, where t is a term fixed to a value v (i.e. $t = v$ in the current store), is that of $P_q[v/x]$, where $[v/x]$ is the operation of (syntactical) replacement of every occurrence of x by v.

Given $q(x) \stackrel{\text{def}}{=} P_q$, we will use $q, qarg$ to denote any two variables not in $fv(P_q)$. Let $\ulcorner x := t \urcorner$ be defined as the process $\sum_v \mathbf{when}\, t = v\, \mathbf{do}\, !\, \mathbf{tell}(x = v)$, i.e., the persistent assignment of t's fixed value to x. Then the ntcc process corresponding to definition of $q(x)$, denoted as $\ulcorner q(x) \stackrel{\text{def}}{=} P_q \urcorner$, is :

$$!\ (\mathbf{when}\ \mathsf{call}(q)\ \mathbf{do}\ \mathbf{local}\, x\ \mathbf{in}\ (\ulcorner x := qarg \urcorner \parallel \ulcorner P_q \urcorner))\,,$$

where $\ulcorner P_q \urcorner$ denotes the process that results from replacing in P_q each $q(t)$ with $\mathbf{tell}(\mathsf{call}(q)) \parallel \mathbf{tell}(qarg = t)$ (thus telling that there is a call of q with argument t). Intuitively, whenever the process q is called with argument $qarg$, the local x is assigned the argument's value so it can be used by q's body $\ulcorner P_q \urcorner$.

We then consider the calls $q(t)$ in other processes. Each such a call is replaced by $\mathbf{local}\, q\ qarg\ \mathbf{in}\ (\ulcorner q(x) \stackrel{\text{def}}{=} P_q \urcorner \parallel \mathbf{tell}(\mathsf{call}(q)) \parallel \mathbf{tell}(qarg = t))$, which we shall denote by $\ulcorner q(t) \urcorner$. The local declarations are needed to avoid interference with other recursive calls.

The above encoding generalizes easily to stratified recursion and to the case of arbitrary number of parameters including the parameterless recursion of tcc considered in [16]. We now show some temporal properties satisfied by the encoding. Next theorem describes the strongest temporal formulae satisfied by $\ulcorner q(t) \urcorner$.

Proposition 2. *Given $\ulcorner q(x) \stackrel{\text{def}}{=} P_q \urcorner$, let B the strongest temporal formula derivable for $\ulcorner P_q \urcorner$. Then the temporal formula*

$$\exists_{q,qarg}(\mathsf{call}(q) \wedge qarg = t \wedge \Box(\mathsf{call}(q) \Rightarrow \exists_x (B \wedge \bigwedge_w (qarg = w \Rightarrow \Box x = w)))$$

is the strongest temporal formula derivable for $\ulcorner q(t) \urcorner$.

The above proposition gives us a proof principle for recursive definitions, i.e., in order to prove that $\ulcorner q(t) \urcorner \vdash A$ it is sufficient to prove that a strongest temporal formula of $\ulcorner q(t) \urcorner$ implies A. The next corollary states a property that one would expect of recursive calls, i.e., if B is satisfied by $q's$ body then $B[v/x]$ should be satisfied by $q(t)$ provided that $t = v$.

Corollary 2. *Given* $\ulcorner q(x) \stackrel{\text{def}}{=} P_q \urcorner$, *suppose that* $q, qarg$ *do not occur free in* B *and* $\ulcorner P_q \urcorner \vdash B$. *Then for all* $v \in Dom$, $\ulcorner q(t) \urcorner \vdash t = v \Rightarrow B[v/x]$.

Cells. Cells provide a basis for the specification and analysis of mutable and persistent data structures. A *cell* can be thought of as a structure that contains a value, and if tested, it yields this value. A *mutable cell* is a cell that can be assigned a new value[1]. We model mutable cells of the form $x:(v)$, which we interpret as a variable x currently fixed to some v.

$$
\begin{array}{lcl}
x\!:(z) & \stackrel{\text{def}}{=} & \textbf{tell}(x = z) \parallel \textbf{unless}\ \mathsf{change}(x)\ \textbf{next}\ x\!:(z) \\
exch_f(x,y) & \stackrel{\text{def}}{=} & \sum_v \textbf{when}\ (x = v)\ \textbf{do}\ (\,\textbf{tell}(\mathsf{change}(x)) \parallel \textbf{tell}(\mathsf{change}(y)) \\
& & \qquad\qquad\qquad \parallel\ \textbf{next}(\ulcorner x\!:(f(v)) \urcorner \parallel \ulcorner y\!:(v) \urcorner)\ ).
\end{array}
$$

Definition $x:(z)$ represents a cell x whose current content is z. The current content of x will be the same in the next time interval unless it is to be changed next (i.e $\mathsf{change}(x)$). Definition $exch_f(x,y)$ represents an exchange operation between the contents of x and y. If v is x's current value then $f(v)$ and v will be the next x and $y's$ values, respectively. In the case of functions that always return the same value (i.e. constants), we will take the liberty of using that value as its symbol. For example, $\ulcorner x:(3) \urcorner \parallel \ulcorner y:(5) \urcorner \parallel \ulcorner exch_7(x,y) \urcorner$ gives us the cells $x:(7)$ and $y:(3)$ in the next time interval.

The following temporal property states the invariant behavior of a cell, i.e., if it satisfies A now, it will satisfy A next unless it is changed.

Proposition 3. *For all* $v \in Dom$, $\ulcorner x:(v) \urcorner \vdash (A \wedge \neg \mathsf{change}(x)) \Rightarrow \circ A$.

Zigzagging. An RCX is a programmable, microcontroller-based LEGO® brick used to create autonomous robotic devices (see e.g., [11]). Zigzagging [8] is a task in which an (RCX-based) robot can go either forward, left, or right but (1) it cannot go forward if its preceding action was to go forward, (2) it cannot turn right if its second-to-last action was to go right, and (3) it cannot turn left if its second-to-last action was to go left.

In order to model this problem, without over-specifying it, we use guarded choice and cells. We use cells act_1 and act_2 to be able to "look back" one and two time units, respectively. We use three distinct $\mathtt{f},\mathrm{r},\mathrm{l} \in Dom - \{0\}$ (standing

[1] A richer notion of cell can be found in ccp based models such as the Oz calculus [19], the π^+ calculus [6], and PiCO [1].

for forward, right and left respectively) and three distinct forward,right,left $\in \mathcal{C}$.

$$
\begin{array}{lll}
\mathit{GoForward} & \stackrel{\text{def}}{=} & \ulcorner \mathit{exch}_{\mathtt{f}}(\mathit{act_1}, \mathit{act_2}) \urcorner \parallel \mathbf{tell}(\mathtt{forward}) \\
\mathit{GoRight} & \stackrel{\text{def}}{=} & \ulcorner \mathit{exch}_{\mathtt{r}}(\mathit{act_1}, \mathit{act_2}) \urcorner \parallel \mathbf{tell}(\mathtt{right}) \\
\mathit{GoLeft} & \stackrel{\text{def}}{=} & \ulcorner \mathit{exch}_{\mathtt{l}}(\mathit{act_1}, \mathit{act_2}) \urcorner \parallel \mathbf{tell}(\mathtt{left}) \\
\mathit{Zigzag} & \stackrel{\text{def}}{=} & (\ \mathbf{when}\,(\mathit{act_1} \neq \mathtt{f})\,\mathbf{do}\,\ulcorner \mathit{GoForward} \urcorner \\
& & +\,\mathbf{when}\,(\mathit{act_2} \neq \mathtt{r})\,\mathbf{do}\,\ulcorner \mathit{GoRight} \urcorner \\
& & +\,\mathbf{when}\,(\mathit{act_2} \neq \mathtt{l})\,\mathbf{do}\,\ulcorner \mathit{GoLeft} \urcorner\ ) \\
& & \parallel\ \mathbf{next}\ \mathit{Zigzag} \\
\mathit{StartZigzag} & \stackrel{\text{def}}{=} & \ulcorner \mathit{act_1}{:}(0) \urcorner \parallel \ulcorner \mathit{act_2}{:}(0) \urcorner \parallel \ulcorner \mathit{Zigzag} \urcorner.
\end{array}
$$

Initially cells $\mathit{act_1}$ and $\mathit{act_2}$ contain neither $\mathtt{f}$,r nor $\mathtt{l}$. Just before a choice is made $\mathit{act_1}$ and $\mathit{act_2}$ contain the previous and the second-to-last taken actions (if any). After a choice is made according to (1), (2) and (3), the choice is recorded in $\mathit{act_1}$ and the previous choice moved to $\mathit{act_2}$. The definitions of the various processes are self-explanatory.

The next temporal property states that the robot chooses to go right and left infinitely often.

Proposition 4. $\ulcorner \mathit{StartZigzag} \urcorner \vdash \Box(\Diamond \mathtt{right} \wedge \Diamond \mathtt{left})$.

Other RCX examples modeled by ntcc includes a crane [20] and a wall-avoiding robot [20].

Value-passing Communication. Value passing plays an important role in several process calculi. Suppose that $x \uparrow (v)$ denotes the action of writing a value (or message) v in channel x which is then kept in the channel for one time unit. We assume that in the same time unit, two different values cannot be written in the same channel. The notation $x \downarrow_{P[y]}$ represents the action of reading, without consuming, the value (if any) in channel x which is then used in P. The variable y, which may occur free in P, is the placeholder for the read value. Several read actions can get the same value if they read the same channel in the same time interval. These basic actions can be defined as $x \uparrow (y) \stackrel{\text{def}}{=} \mathbf{tell}(x = y)$ and $x \downarrow_{P[y]} \stackrel{\text{def}}{=} \sum_v \mathbf{when}\ (x = v)\ \mathbf{do}\,\mathbf{local}\, y\, \mathbf{in}\,(!\,\mathbf{tell}(y = v) \parallel P)$.

Having defined the two basic actions, we can specify different behaviors, e.g., process $!\,(\star_{[0,1]}(x \downarrow_{P[y]}))$ checks "very often" for messages in channel x. Here we illustrate a form of asynchronous broadcasting communication.

$$
\begin{array}{l}
\mathit{SendAsyn}_x(y) \stackrel{\text{def}}{=} \star(\ulcorner x \uparrow (y) \urcorner) \\
\mathit{Waiting}_{Q,x} \stackrel{\text{def}}{=} \mathbf{local}\,\mathit{stop}\,\mathbf{in}\,(\ulcorner x \downarrow_{(Q \parallel \mathbf{tell}(\mathit{stop}=1))[y]} \urcorner \\
\qquad\qquad\qquad\qquad \parallel \mathbf{unless}\,\mathit{stop} = 1\,\mathbf{next}\ \mathit{Waiting}_{Q,x}).
\end{array}
$$

Process $\mathit{SendAsyn}_x(v)$ asynchronously sends value v in channel x. Process $\mathit{Waiting}_{Q,x}$ waits for a value in channel x. Note that, if a process is waiting at the time $\mathit{SendAsyn}_x(v)$ is executed, then it is guaranteed to get the value, while other processes may not get it. This property is expressed by the following result.

Proposition 5. *Suppose that $Q \vdash B$ and $stop \notin fv(Q)$. Then for all $v \in Dom$,*

$$\ulcorner SendAsyn_x(v) \urcorner \parallel \ulcorner Waiting_{Q,x} \urcorner \vdash \Diamond B[v/y].$$

5 Related and Future Work

Our proposal is a strict extension of tcc [16], in the sense that tcc can be encoded in (the restricted-choice subset of) ntcc, while the vice-versa is not possible because tcc does not have constructs to express non-determinism or unbounded finite-delay. In [16] the authors proposed also a proof system for tcc, based on an intuitionistic logic enriched with a next operator. The system, however, is complete only for hiding-free and recursion-free processes. In contrast our system is based on the standard classical temporal logic of [12] and is complete for local-independent choice ntcc processes, hence also for tcc processes. Other extension of tcc, which does not consider non-determinism or unbounded finite-delay, has been proposed in [17]. This extension adds strong pre-emption: the "unless" can trigger activity in the current time interval. In contrast, ntcc can only express weak pre-emption. As argued in [4], in the *specification* of (large) timed systems weak pre-emption often suffices (and non-determinism is crucial). Nevertheless, strong pre-emption is important for reactive systems. In principle, strong pre-emption could be incorporated in ntcc: Semantically one would have to consider assumptions about the future evolutions of the system. As for the logic, one would have to consider a temporal extension of Default Logic [15].

The tccp calculus [4] is the only other proposal for a non-deterministic timed extension of ccp that we know of. One major difference with our approach is that the information about the store is carried through the time units, so the semantic setting is rather different. The notion of time is also different; in tccp each time unit is identified with the time needed to ask and tell information to the store. As for the constructs, unlike ntcc, tccp provides for arbitrary recursion and does not have an operator for specifying (unbounded) finite-delay. A proof system for tccp processes was recently introduced in [3]. The underlying linear temporal logic in [3] can be used for describing input-output behavior while our logic can only be used for the strongest-postcondition. As such the temporal logic of ntcc processes is less expressive than that one underlying the proof system of tccp, but it is also semantically simpler and defined as the standard linear-temporal logic of [12]. This may come in handy when using the Consequence Rule which is also present in [3].

The plan for future research includes the extension of ntcc to a probabilistic model following ideas in [9]. This is justified by the existence of RCX program examples involving stochastic behavior which cannot be faithfully modeled with non-deterministic behavior. In a more practical setting we plan to define a programming language for RCX controllers based on ntcc.

Acknowledgments We are indebted to Mogens Nielsen for having suggested and discussed this work. We thank Andrzej Filinski, Maurizio Gabbrielli, Camilo Rueda, Dale Miller, Vineet Gupta, Radha Jagadeesan and Kim Larsen for helpful comments on various aspects of this work. Thanks goes also to Paulo Oliva, Daniele Varacca, Oliver Moeller, Federico Crazzolara, Giuseppe Milicia and Pawel Sobocinski.

References

1. G. Alvarez, J.F. Diaz, L.O. Quesada, C. Rueda, G. Tamura, F. Valencia, and G. Assayag. Integrating constraints and concurrent objects in musical applications: A calculus and its visual language. *Constraints*, January 2001.
2. G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.
3. F. de Boer, M. Gabbrielli, and M. Chiara. A temporal logic for reasoning about timed concurrent constraint programs. In *TIME 01*. IEEE Press, 2001.
4. F. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Information and Computation*, 1999. To appear.
5. F. S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 19(5):685–725, 1997.
6. J.F. Diaz, C. Rueda, and F. Valencia. A calculus for concurrent processes with constraints. *CLEI Electronic Journal*, 1(2), December 1998.
7. M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. *Theoretical Computer Science*, 183(2):281–315, 1997.
8. J. Fredslund. The assumption architecture. Progress Report, Department of Computer Science, University of Aarhus, November 1999.
9. O. Herescu and C. Palamidessi. Probabilistic asynchronous pi-calculus. *FoSSaCS*, pages 146–160, 2000.
10. I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics. In *Annals of Pure and Applied Logic*, 2000.
11. H. H. Lund and L. Pagliarini. Robot soccer with LEGO mindstorms. *Lecture Notes in Computer Science*, 1604, 1999.
12. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer, 1991.
13. R. Milner. A finite delay operator in synchronous ccs. Technical Report CSR-116-82, University of Edinburgh, 1992.
14. R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, 1999.
15. R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1–2):81–132, April 1980.
16. V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, 4–7 July 1994.
17. V. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5–6):475–520, November–December 1996.

18. V. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages*, pages 333–352, 21–23 January 1991.
19. G. Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.
20. F. Valencia. Reactive constraint programming. Progress Report, BRICS, June 2000. Availabe via http://www.brics.dk/∼fvalenci/publications.html.
21. G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.