A Temporal Concurrent Constraint Programming Calculus

Catuscia Palamidessi
Frank D. Valencia

BRICS Report Series RS-01-20

ISSN 0909-0878 June 2001

A Temporal Concurrent Constraint Programming Calculus

Catuscia Palamidessi Frank D. Valencia

June 2001

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 and asynchronous bounded broadcasting, by specifying temporal requirements such as response and invariance, and by modeling timed systems such as RCX controllers. We present a denotational semantics for modeling the strongest-postcondition behavior of ntcc processes, and, based on this semantics, we develop a proof system for proving linear temporal properties of these processes.

1 Introduction

Research on concurrent constraint programming (ccp) for timed systems has attracted growing interest in the last years. Timed systems often involve *specific domains* (e.g., controllers, databases, reservation systems) and have time-constraints *specifying* their behavior (e.g., the lights must be switched on within the next three seconds). The ccp model enjoys a dual operational and declarative logical view allowing, on the one hand, programs to be expressed using a vocabulary and concepts appropriate to the *specific domain*, and on the other hand, to be read and understood as (logical) *specifications*. An obvious benefit of this view is to provide the developer with one domain specific ccp language suitable for both the *specification* and *implementation* of programs. Indeed, several timed extensions of ccp have been developed in order to provide settings for the programming and specification of timed systems with the declarative flavor of concurrent constraint programming ([31], [30], [8], [12], [13]).

1.1 Concurrent Constraint Programming: the ccp model

Concurrent constraint programming [32] has emerged as a simple but powerful paradigm for concurrency tied to logics. Ccp subsumes and generalizes both concurrent logic programming ([34]) and constraint logic programming ([19]). A fundamental issue in ccp is the *specification of concurrent systems by means of constraints.* A constraint (e.g. $x + y > 10$) represents partial information about certain variables. During the computation, the current state of the system is specified by a set of constraints (*store*). Processes synchronize by *asking* and *telling* information. Whenever a process asks some information not yet entailed by the current store, it blocks, and remains blocked until some other process adds (tells) the requested information to the store.

In the ccp model processes are built by using the basic actions ask and tell, and the operators of parallel composition, hiding, recursion and guarded-choice. Unlike other models of concurrency, without guarded-choice the model is deterministic, namely the result of a finite computation is always the same, independently from the execution order (scheduling) of the parallel components ([33]).

1.2 Reactive Concurrent Constraint Programming: the tcc model

The tcc model ([31]) is a formalism for reactive ccp which combines deterministic ccp with ideas from the Synchronous Languages ([4], [15]). Whenever a tcc process receives a stimulus (partial information) c from the environment, it executes a deterministic ccp process P with c as initial store. In a bounded period of time, P reaches a resting point, and returns the information contained in the final store as a response to the environment. The residual ccp process at the resting point, P', determines the ccp process P'' to be executed in the next time interval. Each stimulus-response interaction between a process and its environment defines a *time unit* (or *time interval*). Since the computation in each time interval is deterministic, tcc is deterministic.

Many interesting temporal constructs can be expressed in tcc. In particular, the **do** P **watching** c construct of ESTEREL [4], which executes P continuously until the signal c is present. In general, tcc allows processes to be "clocked" by other processes, thus allowing meaningful pre-emption constructs.

1.3 A Model of Temporal Concurrent Constraint Programming

Being a model of reactive ccp based on the Synchronous Languages ([4], [15]) (i.e. programs must be determinate and respond immediately to input signals), the tcc model is not meant for the specification of non-deterministic or asynchronous temporal behavior. 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 5), 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 example, uses a nondeterministic construction to help free the programmer from over-specifying a method of solution. As pointed out in [38], 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. Constraints state *what* is to be satisfied but not *how*. Following this line of reasoning, we argue for a formalism for temporal programming where programs and specifications can be given in the same language. For example, we may think of the ccp program **tell**$(x > 5)$ as a specification satisfied by the ccp programs (or refinements) **tell**$(x = 7)$ and **tell**$(x > 11)$.

Moreover, 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 to be modeled faithfully.

In this paper we propose an extension of tcc, which we call the *ntcc* calculus, for temporal concurrent constraint programming. The *ntcc* calculus is obtained by adding *guarded-choice* for modeling non-deterministic behavior and an *unbounded finite-delay* operator for asynchronous behavior. Computation in ntcc progresses as in tcc, except for the non-determinism 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 will illustrate the expressiveness of ntcc by modeling several constructs such as cells, asynchronous bounded broadcasting, response and invariance, that in turn are useful for specifying timed systems. We also illustrate some applications involving RCX$^{\mathrm{TM}}$ controllers.

The declarative nature of ntcc comes to the surface when we consider the denotational characterization of the strongest postcondition observables, as defined in [7] for ccp, and extended to a timed setting. We show that the elegant model based on closure operators, developed in [33] for deterministic ccp, can be extended to a sound model for ntcc without losing its essential simplicity. Under certain conditions on the kind of guarded-choice allowed within the scope of local variables, which we shall call local-independent choice (i.e, either guards without free occurrences of local variables, internal or mutually exclusive choice), we also obtain completeness.

The logical nature of ntcc comes to the surface when we consider its relation with linear temporal logic: We show that all the operators of ntcc correspond to temporal logic constructs like the operators of ccp correspond to (classical) logic constructs ([7]). Following the lines of the proof system proposed in [7] for ccp, we develop a sound system for proving linear temporal properties of ntcc, and we 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 proof system for tcc in [31], whose underlying logic is intuitionistic rather than classical, is complete for hiding (and recursion) free tcc processes only.

We also report on current research on our notion of equality for ntcc: two processes are equivalent iff no context can distinguish them wrt to the input-output behavior. We show the existence of an universal (distinguishing) context and that such an equality is decidable for hiding-free and bounded non-deterministic processes.

The main contributions of this paper can be summarized as follows: (1) a model of temporal concurrent constraint programming that extends the specification power of the tcc model, (2) a denotational semantics capturing the strongest postcondition behavior of a ntcc process, (3) a proof system for proving whether a given ntcc process satisfies a property specified in linear temporal logic, and (4) an study of a natural behavioral equivalence for our calculus.

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.

2.1 Constraint Systems

Concurrent constraint languages are parametrized by a *constraint system.* Basically, a constraint system defines the underlying universe of the particular language. It provides a signature from which syntactically denotable objects in language called *constraints* can be constructed, and an entailment relation specifying interdependencies between such constraints. For our purposes it will suffice to consider the notion of constraint system based on First-Order Predicate Logic, as it was done in [35][1].

Definition 2.1 *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 the set of variables and $\mathcal{S}$ is the set containing the symbols $\dot{\neg}, \dot{\wedge}, \dot{\Rightarrow}, \dot{\exists}$, $\dot{true}$, and *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 $c \dot{\Rightarrow} d$

[1]See [33] for a more general notion of constraints based on Scott's information systems.

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 C for the set of representants of equivalence classes of constraints. For operational reasons we shall require $\vdash$ to be decidable.

2.2 Syntax

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

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

Informally, the intended behavior is the following: The process $\mathbf{tell}(c)$ adds 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 select non-deterministically one of the P_j ($j \in I$) whose corresponding constraint c_j is entailed by the store. Once an alternative is selected the others are precluded, and if none of them can be selected then all of them will be precluded in the next time interval. We omit "$\in I$", if I is unimportant or obvious and we omit "$\sum_{i \in I}$" when I is a singleton set. In case $I = \emptyset$, we write **skip**. We use the symbol "+" to indicate binary summations. Finally, we use $\sum_{i \in I}\ P_i$ as an abbreviation for $\sum_{i \in I} \mathbf{when}\ (\mathit{true})\ \mathbf{do}\ P_i$ (i.e., blind choice).

The process $P \parallel Q$ represents the parallel activation of P and Q. We use $\prod_{i \in I} P_i$, where I is finite, to denote the parallel composition of all P_j, for $j \in I$.

The process $\mathbf{local}\ x\ \mathbf{in}\ P$ 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. We use $\mathbf{local}\ \bar{x}\ \mathbf{in}\ P$ as a shorthand for $\mathbf{local}\ x_1\ \mathbf{in}\ (\mathbf{local}\ x_2\ \mathbf{in}\ (\ldots(\mathbf{local}\ x_n\ \mathbf{in}\ P)\ldots))$, where $\bar{x}$ represents the sequence $x_1 x_2 \ldots x_n$.

The process $\mathbf{next}\ P$ represents the activation of P in the next time interval. 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 $\mathbf{next}$ is repeated n times.

The operator "!" is a delayed version of the replication operator for the $\pi-$calculus ([26]): $!\,P$ 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, so there is no risk of infinite activity within a time interval. The replication operator is the only way of defining infinite behavior through the time intervals. The operator "$\star$" is reminiscent of the finite delay operator for synchronous CCS ([25]) and it allows us to express asynchronous behavior through the time intervals. The process $\star\,P$ represents a finite but unbounded 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.

Note that the bounded versions of $!\,P$ and $\star\,P$ can be derived from the previous constructs. We use $!_I P$ and $\star_I P$, where I is finite, 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.

2.3 Some examples

We now show some examples illustrating specification of temporal behavior in ntcc such as response requirements. Assume that the underlying constraint system includes the predicate symbols in $\{\mathit{Off}, \mathit{TurnOn}, \mathit{OutofOrder}, \mathit{OverHeated}\}$. Consider the "power saving" process

$$!\,(\mathbf{unless}\,(\mathit{Off}\,(\mathit{lights}))\,\mathbf{next}\,\star\,\mathbf{tell}\,(\mathit{Off}\,(\mathit{lights}))).$$

Call it $!\,P$. This process triggers a copy of P each time unit. Thus, the lights are eventually turned off, unless the environment (or another process) tells P that the lights are already turned off. Process P is always active. We may want, however, to specify that the light must be turned off not only eventually but within the next 60 time units. A process specifying this and thus "refining" the previous one would be

$$!\,(\mathbf{unless}\,(\mathit{Off}\,(\mathit{lights}))\,\mathbf{next}\,\star_{[0,60]}\,\mathbf{tell}\,(\mathit{Off}\,(\mathit{lights}))).$$

Finally, we may also want to write an "implementation" of these specifications. For instance, the process

$$!\,(\mathbf{unless}\,(\mathit{Off}\,(\mathit{lights}))\,\mathbf{next}\,\mathbf{tell}\,(\mathit{Off}\,(\mathit{lights})))$$

is one of the possible deterministic processes implementing the above two.

Another example is the specification of (bounded) invariance requirements. Consider the following two processes:

$$\begin{array}{l} !\,(\mathbf{when}\ \ \mathit{OutofOrder}(M)\,\mathbf{do}\,!\,\mathbf{tell}(\dot{\neg}\mathit{TurnOn}(M))) \\ !\,(\mathbf{when}\ \ \mathit{OverHeated}(M)\,\mathbf{do}\,!_{[0,t]}\mathbf{tell}(\dot{\neg}\mathit{TurnOn}(M))) \end{array}$$

The first process repeatedly checks the state of a machine M and, whenever it detects that M is out of order, it tells the other processes that M should not be used anymore. The second process, whenever it detects that M is overheated, tells other processes that M should not be turned on during the next t time units.

2.4 An operational semantics for ntcc

We define now an operational semantics for ntcc which formalizes the intended meaning explained above.

2.4.1 The store and the configurations

Operationally, the current information is represented as a constraint $c \in C$, 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 $\mathbf{local}\ (x, \dot{true})\ \mathbf{in}\ P$.

2.4.2 A structural congruence

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 instance, we have $fv(x = x\dot{\wedge} y > 1) = \{y\}$. For the bound variables, define $bv(c) = \{x \in \mathcal{V} \mid x \text{ occurs in } c\} - fv(c)$. Regarding processes, define $fv(\mathbf{tell}(c)) = fv(c)$, $fv(\sum_i \mathbf{when}\ c_i\ \mathbf{do}\ P_i) = \bigcup_i fv(c_i) \cup fv(P_i)$, and similarly for the bound variables. Further, define $fv(\mathbf{local}\,x\,\mathbf{in}\,P) = fv(P) - \{x\}$ and $bv(\mathbf{local}\,x\,\mathbf{in}\,P) = bv(P) \cup \{x\}$. The other cases are defined inductively in the obvious way.

Definition 2.2 (Structural congruence) *Let $\equiv$ be the smallest congruence relation over processes satisfying the following laws:*

1. $(Proc/_{\equiv}, \|, \mathbf{skip})$ *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. $\mathbf{local}\,x\,\mathbf{in}\,\mathbf{skip} \equiv \mathbf{skip} \qquad \mathbf{local}\,x\,y\,\mathbf{in}\,P \equiv \mathbf{local}\,y\,x\,\mathbf{in}\,P$
5. $\mathbf{local}\,x\,\mathbf{in}\,\mathbf{next}\,P \;\equiv \mathbf{next}(\mathbf{local}\,x\,\mathbf{in}\,P)$
6. $\mathbf{local}\,x\,\mathbf{in}\,(P \parallel Q) \equiv\; P \;\parallel\; \mathbf{local}\,x\,\mathbf{in}\,Q \quad \textit{if} \quad x \notin fv(P)$

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

One interesting property of our calculus is that it admits a notion of standard form:

Definition 2.3 *A process P of the form*

$$\mathbf{local}\ \bar{x}\ \mathbf{in}\ \prod_{i\in[0,n]} \mathbf{next}^i \left(\prod_{j_i} P_{j_i} \parallel \prod_{k_i} \mathbf{unless}\, c_{k_i}\, \mathbf{next}\, Q_{k_i} \parallel \prod_{l_i} !\, Q_{l_i} \parallel \prod_{m_i} \star\, Q_{m_i} \right)$$

is said to be in standard form if each P_{j_i} is a non-empty summation or a tell process, and each Q_{k_i}, Q_{l_i} and Q_{m_i} is itself a standard form.

Proposition 2.4 *Every process P in the original syntax (i.e. with no occurrences of constructs of the form* $\mathbf{local}\,(x, c)\,\mathbf{in}\,P$*) is structurally congruent to a process Q in standard form, and such Q is unique modulo congruence.*

2.4.3 Reduction Relations

The reduction relations $\longrightarrow\ \subseteq \Gamma \times \Gamma$ and $\Longrightarrow\ \subseteq Proc \times C \times C \times Proc$ 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, and PAR are standard. The intuition behind LOC is the following: From the internal point of view of P, the global information about the variable x cannot be observed. Thus, in order to reduce $\langle \textbf{local}\ (x, c)\ \textbf{in}\ P,\ d\rangle$, we should first hide the information about x that d may have. We do this by existentially quantifying x in d. From the external point of view, the internal information produced in c' about x cannot be observed, thus we quantify x in c' in the global store. Additionally, c' becomes the new private store of the process for its future evolutions.

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. Since this is the only way of specifying infinite behavior, it follows that there can be only finitely many internal transitions in one 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 finite 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 $\textbf{next}\,R$ expressions. More precisely:

Definition 2.5 *$F : Proc \to Proc$ is defined as follows:*

$$F(P) = \begin{cases} Q & \textit{if } P = \textbf{next}\ Q \textit{ or } P = \textbf{unless}\ c\ \textbf{next}\ Q \\ F(P_1) \parallel F(P_2) & \textit{if } P = P_1 \parallel P_2 \\ \textbf{local}\ x\ \textbf{in}\ F(Q) & \textit{if } P = \textbf{local}\,(x, c)\ \textbf{in}\ Q \\ \textbf{skip} & \textit{otherwise} \end{cases}$$

Note that both $F(!\,P)$ and $F(\star\,P)$ are defined to be **skip** because neither $!\,P$ nor $\star\,P$ occurs at the top level in a final configuration.

We conclude this section illustrating how processes evolve through the time intervals. An (infinite) sequence of observable transitions

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

can be interpreted as a *stimulus-response interaction* between the process P_1 and an environment. At the time unit i, the environment provides a stimulus c_i and the system 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 by the notation

$$P_1 \xRightarrow{(\alpha,\alpha')}{}^{\omega}$$

A run can alternatively be interpreted as an interaction among the parallel components in the initial system (each component being part of the environment of the others): if $\alpha = \dot{true}.\dot{true}.\dot{true} \ldots$, i.e., the input sequence is empty, then α' can be regarded as a timed observation of such an interaction.

3 Strongest postconditions: denotation and logic for ntcc

In this section we introduce a notion of observables suitable to be represented logically, and we investigate its denotational counterpart.

In the following we use α, α' to represent elements of C^ω and β to represent an element of C^*. Given $c \in C$, $c.\alpha$ 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.

Definition 3.1 (Observables) *1. The input-output (or stimulus-response) relation of a process P is defined as*

$$io(P) = \{(\alpha, \alpha') \mid P \xRightarrow{(\alpha,\alpha')}{}^{\omega}\}$$

2. The quiescent sequences of a process P are defined as

$$sp(P) = \{\alpha \mid P \xRightarrow{(\alpha,\alpha)}{}^{\omega}\}$$

Following [7] we shall refer to $sp(P)$ as the strongest postcondition of P (wrt C^ω) as it satisfies the following:

Proposition 3.2 *$\alpha \in sp(P)$ iff there exists α' such that $P \xRightarrow{(\alpha',\alpha)}{}^{\omega}$.*

3.1 Denotational semantics

We give now a denotational characterization of the strongest postcondition observables of ntcc, following ideas developed in [7] and [31] for the ccp and tcc case, respectively. The presence of non-determinism, however, presents a technical problem to deal with: The observables for the hiding operator cannot be specified compositionally (see [7]). Therefore, we will have to identify a practical fragment for which the semantics is complete wrt our observables.

The denotational semantics is defined as a function $[\![\cdot]\!]$ which associates to each process a set of infinite constraint sequences, namely $[\![\cdot]\!] : \mathit{Proc} \rightarrow \mathcal{P}(C^{\infty})$. The definition of this function is given in Table 2.

Intuitively, $[\![P]\!]$ is meant to capture the quiescent sequences of a process P. For instance, the sequences to which $\mathbf{tell}(c)$ cannot add information 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 $d.\alpha$ is quiescent for it if α is quiescent for P (D5). A sequence is quiescent for $!\, P$ if every suffix of it is quiescent for P (D7). A sequence is quiescent for $\star\, P$ if there is a suffix of it which is quiescent for P (D8). The other rules can be explained analogously.

Remark 3.3 *The ! and the $\star$ operators are dual. In fact, we could have defined*

$$\begin{array}{rcl} [\![!\, P]\!] & = & \nu_X \ \ ([\![P]\!] \cap \{d.\alpha \mid d \in C, \alpha \in X\}) \\ [\![\star\, P]\!] & = & \mu_X \ \ ([\![P]\!] \cup \{d.\alpha \mid d \in C, \alpha \in X\}) \end{array}$$

where ν and μ represent respectively the greatest and the least fix-point operators in the complete lattice $(\mathcal{P}(C^{\infty}), \subseteq)$.

Next theorem states the relation between the denotational semantics of a ntcc process and its strongest postconditions.

Theorem 3.4 (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. As in ccp, in ntcc the converse holds for *restricted choice* processes, namely those ntcc processes in which, for every construct of the form $\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$, the c_i's are pairwise either mutually exclusive or equivalent. Formally, this means that for all $i, j \in I$, if there exists $d \neq \mathit{false}$ such that $d \vdash c_i$ and $d \vdash c_j$, then $c_i = c_j$. Blind-choice processes are a typical case of restricted-choice. The condition required for restricted-choice processes implies *structural confluence* in the sense of [10], namely the outcome of a process does not depend upon the scheduling strategy on its parallel components.

Nevertheless, for ntcc we can show that the converse holds for a larger set of processes which we call *local-independent choice* processes. These are processes in which, for every construct $\sum_{i\in I}$ **when** c_i **do** P_i *occurring within a process* **local** x **in** Q, either (a) $x \notin \bigcup_{i\in I} fv(c_i)$, or (b) the c_i's are pairwise mutually exclusive or equivalent. This fragment is practical since every restricted-choice process is also local-independent choice and, unlike the restricted-choice fragment, its condition does not imply structural congruence. In fact, all the process examples in this paper belong to the local-independent choice fragment but not all of them belong to the restricted choice one (Zigzagging in Section 5).

Theorem 3.5 (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 $S \subseteq C^\infty$ in the complete lattice $(C^\infty, \leq)$.

Theorem 3.6 *If P is a deterministic process, then $(\alpha, \alpha') \in io(P)$ iff $\alpha' = min([\![P]\!] \cap \uparrow \alpha)$, where $\uparrow \alpha = \{\alpha'' | \alpha \leq \alpha''\}$.*

4 A logic for ntcc

In this section we define a linear temporal logic for expressing properties of ntcc processes.

4.1 Syntax

The temporal logic formulae $A, B, ... \in \mathcal{A}$ are defined by the following grammar.

$$A ::= c \mid A \vee A \mid A \wedge A \mid A \Rightarrow A \mid \neg A \mid \exists_x A \mid \circ A \mid \Box A \mid \Diamond A$$

In the above grammar, c denotes an arbitrary constraint. The intended meaning of the other symbols is the following: $\vee$, $\wedge$, $\Rightarrow$, $\neg$ and $\exists_x$ represent temporal logic disjunction, conjunction, implication, negation and existential quantification. These symbols are not to be confused with the logic symbols $\dot{\wedge}$, $\dot{\Rightarrow}$, $\dot{\neg}$ and $\dot{\exists}_x$ of the constraint system. The symbols $\circ$, $\square$, and $\diamond$ denote the temporal operators *next*, *always* and *sometime*.

4.2 Semantics

The standard interpretation structures of linear temporal logic are infinite sequences of states [23]. In the case of ntcc, it is natural to replace states by constraints, and consider therefore as interpretations the elements of C^∞. We say that $\alpha \in C^\infty$ is a model of A, notation $\alpha \models A$, if $\langle \alpha, 1\rangle \models A$, where:

$\langle \alpha, i\rangle \models c$	iff	$\alpha(i) \vdash c$
$\langle \alpha, i\rangle \models \neg A$	iff	$\langle \alpha, i\rangle \not\models A$
$\langle \alpha, i\rangle \models A_1 \vee A_2$	iff	$\langle \alpha, i\rangle \models A_1$ or $\langle \alpha, i\rangle \models A_2$
$\langle \alpha, i\rangle \models A_1 \wedge A_2$	iff	$\langle \alpha, i\rangle \models A_1$ and $\langle \alpha, i\rangle \models A_2$
$\langle \alpha, i\rangle \models A_1 \Rightarrow A_2$	iff	$\langle \alpha, i\rangle \models A_1$ implies $\langle \alpha, i\rangle \models A_2$
$\langle \alpha, i\rangle \models \circ A$	iff	$\langle \alpha, i+1\rangle \models A$
$\langle \alpha, i\rangle \models \square A$	iff	for all $j \geq i$ $\langle \alpha, j\rangle \models A$
$\langle \alpha, i\rangle \models \diamond A$	iff	there exists $j \geq i$ s.t. $\langle \alpha, j\rangle \models A$
$\langle \alpha, i\rangle \models \exists_x A$	iff	there exists $\alpha' \in C^\infty$ s.t. $\dot{\exists}_x \alpha = \dot{\exists}_x \alpha'$ and $\langle \alpha', i\rangle \models A$.

We define $[\![A]\!]$ to be the collection of all models of A. Formally:

$$[\![A]\!] \;=\; \{\alpha \mid \alpha \models A\}$$

4.3 Proving properties of ntcc processes

We are interested in assertions of the form $P \vdash A$, whose intuitive meaning is that the strongest postcondition of P satisfies the property expressed by 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 4.1 *For every ntcc process P and every formula A, $P \vdash A$ holds iff $[\![P]\!] \subseteq [\![A]\!]$ holds.*

Note: the reason why this theorem is called "relative completeness" is because of the side condition in Rule P9 (consequence rule): the proof system is complete modulo the capability of proving the formulae of the form $A \Rightarrow B$ which we need to use in a proof. Proving $A \Rightarrow B$ is known to be decidable for the quantifier-free fragment of linear time temporal formulae ([23]) as well as for some other interesting first-order fragments (see [17]).

From Theorems 4.1, 3.4 and 3.5 we immediately derive the following:

Corollary 4.2
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 [31] 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 5, for discussing properties of our examples.

Definition 4.3 *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 4.4 *The function $stf : Proc \rightarrow \mathcal{A}$ is defined as follows:*

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

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

Proposition 4.5 *For every process P, $stf(P)$ is the strongest temporal formula derivable for P.*

Note that to prove that $P \vdash A$ is sufficient to prove that $stf(P) \Rightarrow A$. 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.

5 Applications

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

Definition 5.1 *Let max be a positive integer number. Define $FD[max]$ as the constraint system whose signature Σ includes symbols in $\{0, succ, +, \times, =\}$ 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.

5.1 Recursion

Often it is convenient to specify behavior by using recursive definitions. In our language we do not have them, but we can show that we can encode a (restricted) form of recursion. Namely, we consider 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 do not want P_q to make infinitely or unboundely many recursive calls of q within the same time interval.

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.

We now show how to encode such a kind of recursion by using replication. 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 $\sum_v \textbf{when}\, t = v\, \textbf{do}\, !\, \textbf{tell}(x = v)$, 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 :

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

where $\ulcorner P_q \urcorner$ denotes the process that results from replacing in P_q each $q(t)$ with $\textbf{tell}(call(q)) \parallel \textbf{tell}(qarg = t)$ (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 $\textbf{local}\, q\ qarg\, \textbf{in}\, (\ulcorner q(x) \stackrel{\text{def}}{=} P_q \urcorner \parallel \textbf{tell}(call(q)) \parallel \textbf{tell}(qarg = t))$, 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 [31]. We now show some temporal properties satisfied by the encoding. Next theorem describes the strongest temporal formulae satisfied by $\ulcorner q(t) \urcorner$.

Proposition 5.2 *Given $\ulcorner q(x) \stackrel{def}{=} P_q \urcorner$, let B the strongest temporal formula derivable for $\ulcorner P_q \urcorner$. Then the temporal formula*

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

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

This theorem expresses the essence of our encoding of recursion in terms of linear temporal logic. It 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 5.3 *Given $\ulcorner q(x) \overset{def}{=} P_q \urcorner$, suppose that $q, qarg$ do not occur free in B and $\ulcorner P_q \urcorner \vdash B$. Then for all $v \in Dom$, $\ulcorner q(t) \urcorner \ \vdash \ t = v \Rightarrow B[v/x]$.*

5.2 Cells

Cells provide a basis for the specification and analysis of mutable and persistent data structures as shown for the π calculus [26]. 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[2]. We model mutable cells of the form $x\colon (v)$, which we interpret as a variable x currently fixed to some v.

$$\begin{array}{lcl} x\colon (z) & \overset{\text{def}}{=} & \textbf{tell}(x = z) \parallel \textbf{unless}\ \mathit{change}(x)\,\textbf{next}\ x\colon (z) \\ f_{\mathit{exch}}(x, y) & \overset{\text{def}}{=} & \sum_v \textbf{when}\ (x = v)\ \textbf{do}\ (\ \textbf{tell}(\mathit{change}(x)) \parallel \textbf{tell}(\mathit{change}(y)) \\ & & \qquad\qquad\qquad\qquad \textbf{next}(\ulcorner x\colon (f(v)) \urcorner \ \parallel \ \ulcorner y\colon (v) \urcorner)\ ) \end{array}$$

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 $\mathit{change}(x)$). Definition $f_{\mathit{exch}}(x, y)$ 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, $\ulcorner x\colon (3) \urcorner \parallel \ulcorner y\colon (5) \urcorner \parallel \ulcorner 7_{\mathit{exch}}(x, y) \urcorner$ gives us the cells $x\colon (7)$ and $y\colon (3)$ 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 5.4 *For all $v \in Dom$, $\ulcorner x\colon (v) \urcorner \ \vdash \ (A \wedge \neg \mathit{change}(x)) \Rightarrow \circ A$*

[2]A richer notion of cell can be found in ccp based models, either as a primitive construct (the Oz calculus [36]) or as a derived construct (π^+ calculus [9], and PiCO [1]).

5.3 Zigzagging

An RCX is a programmable, microcontroller-based LEGO® brick used to create autonomous robotic devices (see [22], [18]). Zigzagging [11] 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 $f, r, l \in Dom - \{0\}$ (standing for forward, right and left respectively) and three distinct $forward, right, left \in C$.

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

Initially cells act_1 and act_2 contain neither f, r nor l. Just before a choice is made act_1 and 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 act_1 and the previous choice moved to 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 5.5 $\ulcorner StartZigzag\urcorner \vdash \Box(\Diamond right \wedge \Diamond left)$

Other RCX examples modeled by ntcc includes a crane [37] and a wall-avoiding robot [37].

5.4 Value-passing Communication

Value passing plays an important role in process calculus. 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 $x \uparrow (y) \stackrel{\text{def}}{=} \textbf{tell}(x = y)$ and $x \downarrow_{P[y]} \stackrel{\text{def}}{=} \sum_v \textbf{when } (x = v) \textbf{ do local } y \textbf{ in } (!\,\textbf{tell}(y = v) \parallel P)$.

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.

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

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 5.6 *Suppose that $Q \vdash B$ and $\mathit{stop} \notin fv(Q)$. Then for all $v \in \mathit{Dom}$,*

$$\ulcorner \mathit{SendAsyn}_x(v) \urcorner \parallel \ulcorner \mathit{Waiting}_{Q,x} \urcorner \vdash \Diamond B[v/y]$$

6 Behavioral Equivalence

We wish to distinguish between the observable behavior of processes P and Q if the distinction can somehow be detected by a process interacting with them. A natural observation is the input-output behavior of P (Definition 3.1). Let $\sim_{io}$ be defined by $P \sim_{io} Q$ iff $io(P) = io(Q)$. Let us consider

$$\begin{array}{lcl} P & = & \textbf{when } \dot{true} \textbf{ do tell}(a) \ + \ \textbf{when } (b) \textbf{ do tell}(c) \\ Q & = & \textbf{when } \dot{true} \textbf{ do tell}(a) \ + \ \textbf{when } (b) \textbf{ do tell}(c) \\ & & + \\ & & \textbf{when } \dot{true} \textbf{ do } (\textbf{tell}(a) \parallel \textbf{when } (b) \textbf{ do tell}(c)) \end{array}$$

Assuming that a, b, c are non equivalent constraints such that $c \vdash b \vdash a$, we can verify that $P \sim_{io} Q$, but $P \parallel R \not\sim_{io} Q \parallel R$ where $R = \textbf{when}\, a\, \textbf{do}\, \textbf{tell}(b)$. This tells us that $\sim_{io}$ is not a *congruence* and that we can distinguish P from Q if we make R to interact with them. Therefore, let $\approx_{io}$ be the corresponding congruence equating processes iff they are (input-output) indistinguishable, i.e., $P \approx_{io} Q$ iff for every process context $C[.]$, $C[P] \sim_{io} C[Q]$. The relation $\approx_{io}$ is our first proper notion of equality for the calculus.

Moreover, let us consider the *language* of a process P, $l(P) = \{\alpha \mid (true.true \ldots, \alpha) \in io(P)\}$, i.e., the set of outputs on the "empty" input sequence. As in the input-output case, we define $\sim_l$ and $\approx_l$ as the corresponding language equivalence and language congruence. Obviously, relation $\sim_l$ is weaker than $\sim_{io}$, however, the corresponding congruences coincide, i.e:

Proposition 6.1 $\sim_{io} \subseteq \sim_l$ *and* $\approx_{io} = \approx_l$

We next investigate the type of contexts needed to verify $P \approx_{io} Q$ and focus on relation $\approx_l$ as it is equivalent to $\approx_{io}$. We first observe that it is enough to consider parallel contexts, i.e.,

Proposition 6.2 $P \approx_l Q$ *iff for all* R, $R \parallel P \sim_l R \parallel Q$.

Furthermore, we show that in ntcc we have the notion of *universal context*, i.e., a context that can distinguish two processes iff they are not language (or input-output) congruent. Recall $\mathcal{C}$ is the set of representatives of equivalent classes of constraints in the constraint system (Σ, Δ). In what follows β ranges over elements of $\mathcal{C}^*$. Let us assume that the signature Σ is extended to a signature Σ' with unary predicates t_β and w_β for each β. These predicates are allowed to occur only in the process contexts $U^S[.]$ defined below. We assume that $\mathcal{C}$ is still defined wrt the original signature so it does not involve any t_β or w_β predicates.

Definition 6.3 *Let* $S \subseteq_{fin} \mathcal{C}$. *Define* $\prec_S \; = \; \{(c, c') \in S \times S \mid c' \vdash c \, and \, c \not\vdash c'\}$ *and* $ic(S) = \{c_1 \ldots c_n \in S^* \mid c_1 \prec_S c_2 \prec_S \ldots \prec_S c_n\}$. *For each* β *let* $W_{c.\beta} = \textbf{when}\, c\, \textbf{do}\, T_\beta$ *and* $T_{c.\beta} = \textbf{tell}(c) \parallel W_\beta$ *where* $T_\epsilon = W_\epsilon = \textbf{skip}$. *The distinguishing context wrt* S, $U^S[.]$ *is defined as*

$$!((\sum_{\beta \in ic(S)} \textbf{tell}(w_\beta) \parallel W_\beta) + (\sum_{\beta \in ic(S)} \textbf{tell}(t_\beta) \parallel T_\beta)) \parallel [.]$$

Proposition 6.4 *Suppose that* $\mathcal{C}$ *is finite. Then* $P \approx_l Q$ *iff* $U^{\mathcal{C}}[P] \sim_l U^{\mathcal{C}}[Q]$.

Thus context $U^{\mathcal{C}}[.]$ is the *universal* context, provided that $\mathcal{C}$ is finite. Roughly speaking, in $U^{\mathcal{C}}[P]$, the distinguishing process in parallel with P, say D, can provide all infinite interactions P can have with other processes. Its selected interaction will be identified with an output $(c_1 \dot{\wedge} d_1).(c_2 \dot{\wedge} d_2).\ldots$ (with $c_i \in \{w_{\beta_i}, t_{\beta_i}\}$) of $U^{\mathcal{C}}[P]$. When interacting with Q, D will be forced to select the same interaction in $U^{\mathcal{C}}[Q]$ to match the output of $U^{\mathcal{C}}[P]$.

Nevertheless, if $\mathcal{C}$ is not finite, we can construct specialized distinguishing contexts for arbitrary processes.

Definition 6.5 *Let $\Lambda \subset_{fin}$ Proc. Define const(Λ) as the set whose elements are* $\dot{true}$, *false and all constraints (module logical equivalence) from the closure under conjunction and existential quantification of the constraints occurring in Λ's processes.*

Proposition 6.6 *Let $\Lambda \subset_{fin}$ Proc. For all $P, Q \in \Lambda$, $P \approx_l Q$ iff $U^{const(\Lambda)}[P] \sim_l U^{const(\Lambda)}[Q]$.*

Therefore $U^{const(\Lambda)}$ is an universal context for Λ's processes. Apart from its theoretical value, the ability of constructing distinguishing contexts for arbitrary processes is important as it can be used for proving decidability results for $\approx_{io}$. Note that in order to verify $P \approx_l Q$ it is sufficient to verify $U^{const(\{P,Q\})}[P] \sim_l U^{const(\{P,Q\})}[Q]$. It turns out that $\sim_l$ is decidable for hiding and unbounded-delay free processes. The languages of these processes can be recognized by automata over infinite objects, more precisely Büchi Automata ([6]).

Proposition 6.7 *$\sim_l$ is decidable for hiding and unbounded-delay free processes.*

Corollary 6.8 *$\approx_l$ and $\approx_{io}$ are decidable for hiding and unbounded-delay free processes.*

7 Related Work and Concluding Remarks

7.1 Related Work

The issue of developing a formalism for timed systems with both a logic and an operational flavor has been considered in several works, particularly in the area of temporal logic programming languages (tpl) ([5], [27],

[3], [24]). These proposals provide the machinery for the direct execution of temporal formulas. Another example is the modal process logic (mpl) of [21], where process constructs are included as connectives and formulae are given operational interpretations. In contrast to tpl, our approach is not based on logic programming but on ccp. Consequently, some of the main advantages of ccp over logic programming can also be claimed for ntcc over tpl, in particular it provides us with a more algebraic view of process combinators. Ntcc and mpl are of a different nature: the former provides linear-time temporal specifications, whereas the latter provides branching-time ones. Being based on tcc, ntcc also provides a programming language while mpl is only a specification language.

The ntcc model was inspired in part by the recent work on real-time modeling of RCX micro-controllers programs ([18]). This work presents a method to synthesize control RCX programs by merging them with control automata. Constraints on the behavior of the control program are given in a temporal logic and then translated into control automata by using the Mona tool ([20]).

The works which are most closely related to our paper are those on tcc (timed ccp, [31]). Our proposal is a strict extension of tcc, 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 [31] the authors proposed also a proof system for tcc, based on an intuitionistic logic enriched with a next operator. The system however is partially complete; namely, it is complete only for hiding-free (and recursion-free) processes. In contrast our system is based on the standard classical temporal logic of ([23]) and is complete for local-indepent choice ntcc processes, hence also for tcc processes.

Other extensions of tcc have been proposed in [13, 14, 30]. None of these, however, consider non-determinism or unbounded finite-delay. In [13] default-tcc processes can evolve continuously as well as discretely. The language in [14] adds to default-tcc random assignments with some given distribution. Finally, the language proposed in [30] adds to tcc the possibility of expressing 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 ([8]), in the *specification* of (large) timed systems weak pre-emption often suffices (and non-determinism is crucial). Nevertheless, strong pre-emption is important for modeling reactive systems. In principle, strong pre-emption could be incorporated in ntcc too: 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 [29])

The tccp calculus ([8]) is the only other proposal for a non-deterministic timed extension of ccp that we know of. As such, tccp provides a declarative language for the specification of (large) timed systems. In fact, interesting examples of specifying such systems are given in ([8]). An elegant denotational account for tccp is presented in [8], although no proof system or direct relation with temporal specifications is given. 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. Also, there is no operator for specifying (unbounded) finite-delay. Like ntcc, the deterministic fragment of tccp can be used to program reactive systems. A store that grows monotonically, however, may be inadequate for the kind of application we have in mind, like RCX micro-controllers.

7.2 Concluding remarks and future work

We introduced ntcc, a model of temporal concurrent constraint programming, and showed examples of its applicability to timed systems (e.g., RCX programs). We illustrated how ntcc can express various temporal requirements (e.g., bounded invariance and eventuality) and other constructs (e.g., cells and value passing processes). We provided a denotational semantics that approximates the notion of strongest postcondition of processes (in the sense of [7]) and identified an important fragment for which such a denotation is complete. We defined a linear temporal (classic) logic following the standard definition of [23], and related it with the denotational semantics. This allowed us to define what it means for a process to satisfy a linear temporal specification. Finally, we defined a (relatively) complete proof system for proving that a process satisfies a linear temporal specification, and we applied it to prove temporal properties of our examples.

Our current research in ntcc includes the study of the decidability of $\approx_{io}$ for arbitrary ntcc processes. In the search of a fully-abstract fixpoint model with respect to this input-output behavior, we found that although ntcc allows countable non-determinism and infinite computations to happen (see [2] and [28] for impossibility results under these conditions), a relatively simple model seems to exist as a result of the particular nature of ntcc.

The plan for future research includes the extension of ntcc to a probabilistic model following ideas in [16]. 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, Kim Larsen, Dale Miller, Maurizio Gabbrielli, Vineet Gupta, Radha Jagadeesan and Prakash Panangaden for helpful comments on various aspects of this work.

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*, jan 2001.

[2] Krzysztof R. Apt and Gordon D. Plotkin. Countable nondeterminism and random assignment. *JACM*, 33:724–767, 1986.

[3] H Barringer, M Fisher, D Gabbay, G Gough, and R Owens. Metatem: An introduction. *Formal Aspects of Computing*, 7(5):533–549, 1995.

[4] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[5] Christoph Brzoska. Temporal logic programming and its relation to constraint logic programming. In Kazunori Saraswat, Vijay; Ueda, editor, *Proceedings of the 1991 International Symposium on Logic Programming (ISLP'91)*, pages 661–677, San Diego, CA, October 1991. MIT Press.

[6] J. R. Buchi. On a decision method in restricted second order arithmetic. In *Proc. Int. Cong. on Logic, Methodology, and Philosophy of Science*, pages 1–11. Stanford University Press, 1962.

[7] 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.

[8] Frank de Boer, Maurizio Gabbrielli, and Maria Chiara Meo. A timed concurrent constraint language. *Information and Computation*, 1999. To appear.

[9] Juan Francisco Diaz, Camilo Rueda, and Frank Valencia. A calculus for concurrent processes with constraints. *CLEI Electronic Journal*, 1(2), December 1998.

[10] M. Falaschi, M. Gabbrielli, K. Marriott, and C. Palamidessi. Confluence in concurrent constraint programming. *Theoretical Computer Science*, 183(2):281–315, 1997.

[11] Jakob Fredslund. The assumption architecture, November 1999. BRICS, Progress Report.

[12] V. Gupta, R. Jagadeesan, and V. A. Saraswat. Computing with continuous change. *Science of Computer Programming*, 30(1–2):3–49, January 1998.

[13] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Models for concurrent constraint programming. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 66–83, Pisa, Italy, 26–29 August 1996. Springer-Verlag.

[14] Vineet Gupta, Radha Jagadeesan, and Vijay Saraswat. Probabilistic concurrent constraint programming. In Antoni Mazurkiewicz and Józef Winkowski, editors, *CONCUR '97: Concurrency Theory, 8th International Conference*, volume 1243 of *Lecture Notes in Computer Science*, pages 243–257, Warsaw, Poland, 1–4 July 1997. Springer-Verlag.

[15] N. Halbwachs. Synchronous programming of reactive systems. *Lecture Notes in Computer Science*, 1427:1–16, 1998.

[16] O. Herescu and C. Palamidessi. Probabilistic asynchronous pi-calculus. *FoSSaCS*, pages 146–160, 2000.

[17] I. Hodkinson, F. Wolter, and M. Zakharyaschev. Decidable fragments of first-order temporal logics, 2000.

[18] Thomas S. Hune and Anders B. Sandholm. A case study on using automata in control synthesis. *LNCS-1783*, pages 349–362, 2000.

[19] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 111–119. ACM, ACM Press, 1987.

[20] Nils Klarlund and Anders Moller. *MONA Version 1.3 User Manual.* BRICS, jun 1998.

[21] Kim G. Larsen and Bent Thomsen. A modal process logic. In *Proceedings, Third Annual Symposium on Logic in Computer Science*, pages 203–210, Edinburgh, Scotland, 5–8 July 1988. IEEE Computer Society.

[22] H. H. Lund and L. Pagliarini. Robot soccer with LEGO mindstorms. *Lecture Notes in Computer Science*, 1604, 1999.

[23] Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer, 1991.

[24] S. Merz. Efficiently executable temporal logic programs. *Lecture Notes in Computer Science*, 897, 1995.

[25] R. Milner. A finite delay operator in synchronous ccs. Technical Report CSR-116-82, University of Edinburgh, 1992.

[26] R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, 1999.

[27] B. Moszkowski. *Executing Temporal Logic Programs.* Cambridge University Press, Cambridge, 1986.

[28] Sven-Olof Nyström. There is no fully abstract fixpoint semantics for non-deterministic languages with infinite computations. *IPL*, 60(6):289–293, 1996.

[29] R. Reiter. A logic for default reasoning. *Artificial Intelligence*, 13(1–2):81–132, April 1980.

[30] V. Saraswat, R. Jagadeesan, and V. Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5–6):475–520, November–December 1996.

[31] Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. Foundations of timed concurrent constraint programming. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[32] Vijay A. Saraswat. *Concurrent Constraint Programming.* ACM Doctoral Dissertation Awards: Logic Programming. The MIT Press, Cambridge, MA, 1993.

[33] Vijay A. Saraswat, Martin Rinard, and Prakash Panangaden. The semantic foundations of concurrent constraint programming. In ACM, editor, *POPL '91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages, January 21–23, 1991, Orlando, FL*, pages 333–352, New York, NY, USA, 1991. ACM Press.

[34] E. Shapiro. The Family of Concurrent Logic Programming Languages. *Computing Surveys*, 21(3):413–510, 1990.

[35] Gert Smolka. A Foundation for Concurrent Constraint Programming. In *Constraints in Computational Logics*, volume 845 of *Lecture Notes in Computer Science*, Munich, Germany, September 1994. Invited Talk.

[36] Gert 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.

[37] Frank D. Valencia. Reactive constraint programming, June 2000. BRICS Progress Report - availabe via http://www.brics.dk/~fvalenci/publications.html.

[38] G. Winskel. *The Formal Semantics of Programming Languages.* The MIT Press, 1993.

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	$\dfrac{\langle P, c\rangle \;\longrightarrow^*\; \langle Q, d\rangle \;\not\longrightarrow}{P \;\xRightarrow{(c,d)}\; F(Q)}$

Table 1: An operational semantics for ntcc. The function F, used in OBS, is given in Definition 2.5

D1	$[\![\mathbf{tell}(c)]\!] \;=\; \{d.\alpha \mid d \vdash c, \alpha \in C^{\infty}\}$
D2	$[\![\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i]\!] \;=\; \bigcup_{i\in I}\{d.\alpha \mid d \vdash c_i, d.\alpha \in [\![P_i]\!]\} \cup \bigcap_{i\in I}\{d.\alpha \mid d \not\vdash c_i, d.\alpha \in C^{\infty}\}$
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	$[\![\mathbf{next}\ P]\!] \;=\; \{d.\alpha \mid d \in C, \alpha \in [\![P]\!]\}$
D6	$[\![\mathbf{unless}\ c\ \mathbf{next}\ P]\!] \;=\; \{d.\alpha \mid d \vdash c, \alpha \in C^{\infty}\} \cup \{d.\alpha \mid d \not\vdash c, \alpha \in [\![P]\!]\}$
D7	$[\![!\,P]\!] \;=\; \{\alpha \mid \forall \beta \in C^{*}, \alpha' \in C^{\infty} \text{ s.t. } \alpha = \beta.\alpha',\ \text{we have } \alpha' \in [\![P]\!]\}$
D8	$[\![\star\, P]\!] \;=\; \{\beta.\alpha \mid \beta \in C^{*}, \alpha \in [\![P]\!]\}$

Table 2: Denotational semantics of ntcc

P1	$\mathbf{tell}(c) \vdash c$
P2	$\dfrac{\forall i \in I \quad P_i \vdash A_i}{\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i \vdash \bigvee_{i\in I}(c_i \wedge A_i) \vee \bigwedge_{i\in I} \neg c_i}$
P3	$\dfrac{P \vdash A \qquad Q \vdash B}{P \parallel Q \vdash A \wedge B}$
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 $A \Rightarrow B$

Table 3: A proof system for proving linear temporal properties of ntcc processes