Nordic Journal of Computing

TEMPORAL CONCURRENT CONSTRAINT PROGRAMMING: DENOTATION, LOGIC AND APPLICATIONS

MOGENS NIELSEN *
BRICS, University of Aarhus, Denmark
mn@brics.dk

CATUSCIA PALAMIDESSI †
Penn State University, USA
catuscia@cse.psu.edu

FRANK D. VALENCIA *
BRICS, University of Aarhus, Denmark
fvalenci@brics.dk

Abstract. The tcc model is a formalism for reactive concurrent constraint programming. We present a model of *temporal concurrent constraint programming* which adds to tcc the capability of modeling asynchronous and nondeterministic timed behavior. We call this tcc extension the *ntcc calculus*. We also give 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. The expressiveness of ntcc is illustrated by modeling cells, timed systems such as RCX controllers, multi-agent systems such as the Predator/Prey game, and musical applications such as generation of rhythms patterns and controlled improvisation.

CR Classification: F3.1, F3.2, D1.3, D3.3

Key words: Temporal Concurrent Constraint Programming, Calculi for Concurrency, Program Specification, Timed Systems, Reactive Systems, Multi-Agent Systems, Musical Applications.

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

*The contribution of Mogens Nielsen and Frank D. Valencia to this research has been supported by Research in Computer Science, Centre of the Danish National Research Foundation.

†The contribution of Catuscia Palamidessi to this research has been supported by the NSF-POWRE grant EIA-0074909

Received Dec 2001. Revised April 2002.

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 (Saraswat *et al.* [1994, 1996], de Boer *et al.* [2000], Gupta *et al.* [1998]).

1.1 Concurrent Constraint Programming: the ccp model

Concurrent constraint programming [Saraswat 1993] has emerged as a simple but powerful paradigm for concurrency tied to logics. Ccp extends and subsumes both concurrent logic programming [Shapiro 1990] and constraint logic programming [Jaffar and Lassez 1987]. 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*). Process can change the state of the system by *telling* information to the store (i.e., adding constraints to the store), and synchronize by *asking* information to the store (i.e., determining whether a given constraint can be inferred from the store).

In the ccp model processes are built by using the basic operations ask and tell, and the operators of parallel composition, hiding, guarded-choice (the guards being constraints), and recursion. 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 [Saraswat *et al.* 1991].

1.2 Reactive Concurrent Constraint Programming: the tcc model

The tcc model [Saraswat *et al.* 1994] is a formalism for reactive ccp which combines deterministic ccp with ideas from the Synchronous Languages ([Berry and Gonthier 1992], [Halbwachs 1998]).

In tcc time is conceptually divided into *discrete intervals (or time units)*. Intuitively, in a particular timed interval, a *deterministic* ccp process P receives a stimulus (i.e. piece of information represented as a constraint) c 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 d. The resting point also determines a residual process Q, which is then executed in the next time interval.

The tcc model extends the deterministic ccp model with two temporal operators: $\mathbf{next}\,P$ and $\mathbf{unless}\ c\ \mathbf{next}\,P$. The *next* operator specifies a one time unit delay for the execution of P. The *unless* operator is similar, but the delay is carried only if the information represented by its guard (i.e, the constraint c) cannot be inferred from the store during the current time interval. Many interesting temporal constructs can then be derived in tcc. In particular, the **do** P **watching** c construct of ESTEREL [Berry and Gonthier 1992], 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 (i.e. programs must be determinate and respond immediately to input signals), the tcc model is not meant for modeling nondeterministic 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 7), in which a robot can unpredictably choose its next move, is an example where nondeterminism is useful.

In general, a benefit of allowing the specification of nondeterministic 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 by Winskel [1993], 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*.

Furthermore, a very important benefit of allowing the specification of nondeterministic (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 nondeterminism 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 nondeterministic behavior and an *unbounded finite-delay* operator for asynchronous behavior. Computation in ntcc progresses as in tcc, except for the nondeterminism 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.

The declarative nature of ntcc comes to the surface when we consider the denotational characterization of the strongest postcondition observables, as defined by de Boer *et al.* [1997] for ccp, and extended to a timed setting. We show that the elegant model based on closure operators, developed by

Saraswat *et al.* [1991] for deterministic ccp, can be extended to a sound model for ntcc without losing its essential simplicity. We also obtain completeness for a class of processes that we shall call *locally independent*. These are the processes in which the choice and unless constructs contain no bound variables in their guards. It turns out that the local-independence condition for the choice operator strictly subsumes the so-called *restricted choice* condition considered by [Falaschi *et al.* 1997]. Restricted choice means that in every choice the guards are either pairwise mutually exclusive or equal – e.g., *blind (or internal) choice* falls into this category.

The logical nature of ntcc comes to the surface when we consider its relationship 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. Following the lines of the proof (or inference) system proposed in [de Boer *et al.* 1997] 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 locally independent processes. Our system is then complete for tcc as well, since every tcc process falls into the category of locally independent ntcc processes. The proof system for tcc in [Saraswat *et al.* 1994], whose underlying logic is intuitionistic rather than classical, is complete for hiding (and recursion) free tcc processes only.

Furthermore, we illustrate the expressive power of *ntcc* by modeling constructs such as cells and some applications involving timed systems (RCX$^{\text{TM}}$ controllers), multi-agent systems (the Predator/Prey game), and musical applications (generation of rhythms patterns and controlled improvisation).

The main contributions of this paper can be summarized as follows:

(1) a model of temporal concurrent constraint programming that extends the specification and modeling capabilities of tcc,

(2) a denotational semantics capturing the strongest postcondition behavior of a ntcc process,

(3) an inference system for proving whether a given ntcc process satisfies a property specified in a linear temporal logic, and

(4) applications of temporal concurrent constraint programming.

A preliminary version of this paper, without proofs or a detailed treatment of items (2) and (3), was published as [Palamidessi and Valencia 2001]. Here we shall study in full detail items (1-3), and regarding item (4), we shall illustrate new applications of the calculus.

2. Syntax and Intuitive Behavior

In this section we present the syntax of the various ntcc constructs and their intuitive behavior. First we recall the notion of constraint system which is central to concurrent constraint programming.

2.1 Constraint Systems

The ntcc processes are parametric in a *constraint system.* A constraint system provides a *signature* from which syntactically denotable objects called *constraints* can be constructed and an *entailment relation* $\vdash$ specifying interdependencies between these constraints.

A constraint represents a piece of information (or *partial information*) upon which processes may act. For instance, processes modeling temperature controllers may have to deal with partial information such as $42 < \texttt{tsensor} < 100$ expressing that the sensor registers an *unknown (or not precisely determined)* temperature value between 42 and 100. The interdependency $c \vdash d$ expresses that the information specified by d follows from the information specified by c, e.g. $(\texttt{tsensor} > 42) \vdash (\texttt{tsensor} > 0)$.

We can set up the notion of constraint system by using First-Order Logic as it was done in [Niehren and Smolka 1994]. Let us suppose that Σ is a signature (i.e., a set of constant, functions and predicate symbols) and that Δ is a consistent first-order theory over Σ (i.e., a set of sentences over Σ having at least one model). Constraints can be thought of as first-order formulae over Σ. We can then decree that $c \vdash d$ if the implication $c \Rightarrow d$ is valid in Δ. This gives us a simple and general formalization of the notion of constraint system as a pair (Σ, Δ).

Definition 2.1. (Constraint System) *A* constraint system *is a pair* (Σ, Δ) *where* Σ *is a signature (i.e., a set of constants, functions and predicate symbols) and* Δ *is a consistent first-order theory over* Σ *(i.e., a set of first-order sentences over* Σ *having at least one model).*

Given a constraint system (Σ, Δ), let $\mathcal{L}$ be the underlying first-order language $(\Sigma, \mathcal{V}, \mathcal{S})$, where $\mathcal{V}$ is a countable set of variables and $\mathcal{S}$ is the set containing the symbols $\neg, \wedge, \Rightarrow, \exists, \texttt{true}$ and $\texttt{false}$ which denote logical negation, conjunction, implication, existential quantification, and the always true and always false predicates, respectively. *Constraints,* denoted by $c, d, \ldots$ 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. For operational reasons, we shall require $\vdash$ to be decidable. We say that c is equivalent to d, written $c \approx d$, iff $c \vdash d$ and $d \vdash c$. Henceforward $\mathcal{C}$ denotes the set of constraints modulo $\approx$ in (Σ, Δ).

An alternative formalization of the notion of constraint system is given in [Saraswat *et al.* 1991] by using Scott's information system without consistency structure.

2.2 Process Syntax

The process syntax of ntcc is parametric in an underlying constraint system (Σ, Δ).

Definition 2.2. (Syntax) *Processes P, Q, $\ldots \in Proc$ are built from constraints $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system* (Σ, Δ) *by the following syntax.*

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

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 $\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}$ where I is a finite set of indexes, represents a process that, in the current time interval, must nondeterministically choose one of the P_j $(j \in I)$ whose corresponding guard 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 shall often write $\mathbf{when}\ c_{i_1}\ \mathbf{do}\ P_{i_1} + \ldots + \mathbf{when}\ c_{i_n}\ \mathbf{do}\ P_{i_n}$, the other of the terms being insignificant, if $I = \{i_1, \ldots, i_n\}$ and, if no ambiguity arises, omit the "$\mathbf{when}\ c\ \mathbf{do}$" when $c = \mathtt{true}$. The "blind-choice" process $\sum_{i \in I} \mathbf{when}\ \mathtt{true}\ \mathbf{do}\ P_i$, for example, can be written as $\sum_{i \in I}\ P_i$. We shall omit the "$\sum_{i \in I}$" when I is a singleton. We use **skip** as an abbreviation of the empty summation $\sum_{i \in \emptyset} P_i$.

Process $P \parallel Q$ represents the parallel composition of P and Q. In one time unit P and Q operate concurrently, communicating through the store. We shall use $\prod_{i \in I} P_i$, where $I = \{i_1, \ldots, i_n\}$, to denote the parallel composition $((\ldots (P_{i_1} \parallel P_{i_2}) \parallel \ldots) \parallel P_{i_{n-1}}) \parallel P_{i_n}$.

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 $\mathbf{local}\ x_1 x_2 \ldots x_n\ \mathbf{in}\, P$ as an abbreviation of $\mathbf{local}\ x_1\ \mathbf{in}\,(\mathbf{local}\ x_2\ \mathbf{in}\,(\ldots(\mathbf{local}\ x_n\ \mathbf{in}\, P)\ldots))$.

The only move of $\mathbf{next}\, P$ is a unit-delay for the activation of P. Process P will then be activated in the next time interval in some store which may be unrelated to the one in the current time interval. The process $\mathbf{unless}\ c\ \mathbf{next}\, P$ is similar, but P will be activated only if c cannot be eventually inferred from the information in the store during the current time interval. Notice that $\mathbf{unless}\ c\ \mathbf{next}\, P$ is not the same as $\mathbf{when}\ \neg c\ \mathbf{do}\ \mathbf{next}\, P$ since it could very well be the case that neither $\neg c$ nor c can be inferred from the information in the store. Notice also that $Q = \mathbf{unless}\ \mathtt{false}\ \mathbf{next}\, P$ is not the same as $R = \mathbf{next}\, P$ since unlike Q, even if the store contains $\mathtt{false}$, R will still activate P in the next time interval (and the store in the next time interval may not contain $\mathtt{false}$). We shall 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 "$\star$" corresponds to the unbounded but finite delay operator ϵ for synchronous CCS [Milner 1992] and it allows us to express asynchronous behavior through the time intervals. Intuitively, process $\star\, P$ represents $P + \mathbf{next}\, P + \mathbf{next}^2 P + \ldots$, i.e., an arbitrary long but finite delay for the activation of P.

The operator "!" is a delayed version of the replication operator for the $\pi-$calculus [Milner 1999]: $!\,P$ 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.

2.2.1 Derived Operators.

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 [Milner 1992]. 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 $P \mid Q$).

Note that the bounded versions of $!\,P$ and $\star\,P$ can be derived from the previous constructs. We shall use $!_I P$ and $\star_I P$, where I is an interval of the natural numbers, as abbreviations for $\prod_{i \in I} \mathbf{next}^i P$ and $\sum_{i \in I} \mathbf{next}^i P$, respectively. Intuitively, $\star_{[m,n]} P$ 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 shall give some examples illustrating the specification of temporal behavior in ntcc such as response and invariant requirements. Let us assume that the underlying constraint system includes the predicate symbols in $\{\texttt{Off}, \texttt{TurnOn}, \texttt{OutofOrder}, \texttt{OverHeated}\}$.

EXAMPLE 2.1. (POWER SAVER) Consider the "power saver" process

$$!P \;=\; !\,(\mathbf{unless}\;\; \texttt{LightsOff}\;\; \mathbf{next}\; \star\, \mathbf{tell}(\texttt{LightsOff})).$$

This process triggers a copy of P each time unit. Thus, the lights are eventually turned off, unless the environment or other process tells P that the lights are already turned off. 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

$$!\,(\mathbf{unless}\;\; \texttt{LightsOff}\;\; \mathbf{next}\;\; \star_{[0,60]}\, \mathbf{tell}(\texttt{LightsOff})).$$

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

$$!\,(\mathbf{unless}\;\; \texttt{LightsOff}\;\; \mathbf{next}\,\mathbf{tell}(\texttt{LightsOff}))$$

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

Another example is the specification of (bounded) invariance requirements.

EXAMPLE 2.2. (MACHINES) Consider the following two processes:

$$! (\textbf{when } \texttt{OutofOrder}(M) \textbf{ do} ! \textbf{tell}(\neg \texttt{TurnOn}(M)))$$
$$! (\textbf{when } \texttt{OverHeated}(M) \textbf{ do} !_{[0,t]} \textbf{tell}(\neg \texttt{TurnOn}(M)))$$

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.

3. Operational Semantics

In this section we shall make precise the intuitive process description given in the previous sections. We shall give meaning to the ntcc processes by means of an operational semantics. The semantics, which is inspired by work on the π-calculus, provides *internal* and *external* transitions describing process evolutions. The internal transitions describe evolutions within a time unit and thus they are regarded as being unobservable. In contrast, the external transitions are regarded as being observable as they describe evolution across the time units. We shall introduce some relevant notions of observable process behavior based on the external transitions. Such notions induce behavioral process equivalences that abstract from internal transitions and thus from internal behavior.

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 $\langle P, c\rangle$. Following standard lines, we shall extend the syntax with a construct $\textbf{local}\,(x, d)\,\textbf{in}\,P$, which represents the evolution of a process of the form $\textbf{local}\,x\,\textbf{in}\,Q$, where d is the local information (or store) produced during this evolution. Initially d is “empty”, so we regard $\textbf{local}\,x\,\textbf{in}\,P$ as $\textbf{local}\ (x, \texttt{true})\ \textbf{in}\ P$.

We now introduce a notion of free variables invariant wrt the equivalence on constraints. We define the set of “relevant” *free variables* of c as $fv(c) = \{x \in \mathcal{V} \mid \exists_x c \not\approx c\}$. For example, $fv(x = y) = \{x, y\}$ but $fv(x = x) = fv(\texttt{true}) = \emptyset$. The complementary notion is that of bound variables. We therefore define the set of *bound variables* of a constraint c as $bv(c) = \{x \in \mathcal{V} \mid x \text{ occurs in } c\} - fv(c)$. We next extend these definitions to processes.

DEFINITION 3.1. (FREE AND BOUND VARIABLES) *The set of free variables of a processes P, $fv(P)$, is defined inductively as follows:*
$fv(\textbf{tell}(c)) = fv(c)$,
$fv(\sum_i \textbf{when } c_i \textbf{ do } P_i) = \bigcup_i fv(c_i) \cup fv(P_i)$,
$fv(\textbf{local}\,(x, c)\,\textbf{in}\,P) = (fv(P) \cup fv(c)) - \{x\}$
$fv(\textbf{unless } c \textbf{ next} P) = fv(c) \cup fv(P)$
$fv(\textbf{next} P) = fv(!P) = fv(\star P) = fv(P)$.

A variable x occurring in P is bound *in P iff $x \notin fv(P)$. The set of bound variables of P is denoted by $bv(P)$.*

As usual to make the transition system simpler several expressions in the language are identified with the relation $\equiv$ defined below.

DEFINITION 3.2. (STRUCTURAL CONGRUENCE) *Let $\equiv$ be the smallest congruence over processes satisfying the following axioms:*

(1) $P \parallel \mathbf{skip} \equiv P,\ P \parallel Q \equiv Q \parallel P,\ P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R.$

(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) $\mathbf{local}\, x\, \mathbf{in}\, (P \parallel Q) \equiv P \parallel \mathbf{local}\, x\, \mathbf{in}\, Q \quad \textit{if} \quad x \notin fv(P).$

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

The structural congruence describes irrelevant syntactic aspects of processes in a simple and intuitive way. Notice, for example, that we did not consider the extended processes **local** (x, c) **in** P in the axioms of $\equiv$. The axioms for such processes would require some reasoning about the operational role of local stores, thus adding unnecessary complexity to the definition. Furthermore, as we shall see in the next section, these extended local processes, which were introduced merely for operational reasons, are used only in internal (and thus unobservable) transitions.

3.1 Reduction Relations

The operational semantics will be given in terms of the reduction relations $\longrightarrow$ and $\Longrightarrow$ given by the rules in Table 3.1. The *internal* transition

$$\langle P, d\rangle \longrightarrow \left\langle P', d'\right\rangle$$

should be read as " P with store d reduces, in one internal step, to P with store d' ". We then say that P' is an *internal evolution* of P.

The *observable or external* transition

$$P \xRightarrow{(c,d)} P'$$

should be read as "P on input c from the environment, reduces in one time unit to P' and outputs d to the environment". We then say that P' is an *(observable) evolution* of P.

Intuitively, the above observable reduction is obtained from a sequence of internal reduction starting in P with initial store c and terminating in a process Q with store d. Process P', which is the process to be executed in the next time, is obtained by removing from Q what was meant to be executed only during the current time interval. The store d is not automatically transferred to the next time unit. If needed, information in d can be transfered to next time unit by P itself.

Let us now describe the rules in Table 3.1. Rule TELL, SUM, PAR and UNL are self-explanatory. Instead, we shall dwell at length on the description of Rule LOC as it may seem somewhat complex. Let us consider the process

$$Q = \textbf{local}\ (x, c)\ \textbf{in}\, P$$

in Rule LOC. The global store is d plus some information about the local store c. We distinguish between the *external* (corresponding to Q) and the *internal* point of view (corresponding to P). From the external point of view the internal information about x, possibly appearing in c, cannot be observed by Q. Hence, x in c should be hidden in the global store. We do this by existentially quantifying x in c. Similarly, from the internal point of view, the information about x, possibly appearing in the "global" store d, cannot be observed by P. Therefore, before reducing P we should first hide the information about x that Q may have in d. Now, the information about x that a reduction of P may produce (i.e., c') cannot be observed from the external point of view. Thus, we hide it by existentially quantifying x in c' before adding it to the global store corresponding to the evolution of Q. Furthermore, we should make c' the new private store of the evolution of P for its future reductions.

We now return to describing the rules. Rule STAR says that $\star P$ triggers P in some time interval (either in the current one or in a future one). Notice that this rule generates unbounded nondeterminism (or infinite branching) as any arbitrary $n \geq 0$ can be chosen for the reduction. Rule REP specifies that the process $!\,P$ produces a copy P at the current time unit, and then persists in the next time unit. Notice that there is no risk of infinite behavior within a time unit as we delay one time unit the only way of specifying infinite behavior. Rule STR simply says that structurally congruent configurations have the same reductions.

Rule OBS says that an observable transition from P labeled by (c, d) is obtained by performing a sequence of internal transitions from $\langle P,\, c\rangle$ till we reach a configuration $\langle Q, d\rangle$ where Q is a process that cannot proceed anymore. The residual process R to be executed in the next time interval is equivalent to $F(Q)$ ("future" of Q), which is obtained by removing from Q summations that did not triggered activity within 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 3.3. (FUTURE FUNCTION) *Let $F : Proc \to Proc$ be defined by*

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

REMARK 3.1. Function F could have been defined as a partial function since whenever we need to apply F to a P (Rule OBS in Table 3.1), all $\mathbf{tell}(c)$, $!\,Q$ and $\star Q$ in P will occur within a “next” or “unless” expression.

EXAMPLE 3.1. (ILLUSTRATING REDUCTION) Let us consider the process $!P$ in the power-saver example (Example 2.1). Let $c = \texttt{LightsOff}$ and $R = \mathbf{tell}(c)$. One can verify that for any $n > 0$ the following is a valid sequence of observable transitions starting with $Q = !P \parallel \mathbf{next}\ R$.

$$Q \xRightarrow{(c,c)} !P \parallel R \xRightarrow{(\texttt{true},c)} !P \xRightarrow{(\texttt{true},\texttt{true})} !P \parallel \star R \xRightarrow{(\texttt{true},\texttt{true})} !P \parallel \mathbf{next}^n R$$
$$\xRightarrow{(\texttt{true},\texttt{true})} !P \parallel \mathbf{next}^{n-1} R \xRightarrow{(\texttt{true},\texttt{true})} \dots \xRightarrow{(\texttt{true},\texttt{true})} !P \parallel R \xRightarrow{(\texttt{true},c)} !P.$$

In the first time unit (or time interval) the environment tells c (i.e., c is given as input to Q) thus $!P$ does not trigger its $\star R$ (i.e., the $\star R$ occurring within $!P$). In the second time unit the environment does not tell c but R does, thus $!P$ does not trigger its $\star R$. In the third time unit neither the environment nor other process tell c, thus $!P$ triggers its $\star R$. In the fourth time unit $\star R$ triggers the next process which n time units later executes P. Thus in the $5 + n-$th time unit R tells c.

4. Observable Behavior

In this section we introduce some notions of what an observer (e.g., ourselves) can see from a process behavior. We shall refer to such notions as *process observations*. We assume that what happens within a time unit cannot be directly observed, and thus we abstract from internal transitions. The ntcc calculus makes easy to focus on the observation of input-outputs events in which a given process engages and the order in which they occur.

Henceforth $\mathcal{C}^\omega$ and $\mathcal{C}^*$ denote the set of infinite and finite sequences, respectively, of constraints in $\mathcal{C}$. We shall use α, α' to represent elements of $\mathcal{C}^\omega$ and β to represent an element of $\mathcal{C}^*$. Furthermore, we shall use $\beta.\alpha$ to represent the concatenation of β and α.

$$\text{TELL} \quad \frac{}{\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d \wedge c\rangle}$$

$$\text{SUM} \quad \frac{d \vdash c_j \quad j \in I}{\left\langle \sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i, d\right\rangle \longrightarrow \langle P_j, d\rangle}$$

$$\text{PAR} \quad \frac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle}$$

$$\text{UNL} \quad \frac{d \vdash c}{\langle \mathbf{unless}\ c\ \mathbf{next}\, P, d\rangle \longrightarrow \langle \mathbf{skip}, d\rangle}$$

$$\text{LOC} \quad \frac{\langle P, c \wedge (\exists_x d)\rangle \longrightarrow \langle P', c' \wedge (\exists_x d)\rangle}{\langle \mathbf{local}\ (x, c)\ \mathbf{in}\, P, d \wedge \exists_x c\rangle \longrightarrow \langle \mathbf{local}\ (x, c')\ \mathbf{in}\, P', d \wedge \exists_x c'\rangle}$$

$$\text{REP} \quad \frac{}{\langle !\, P, d\rangle \longrightarrow \langle P \parallel \mathbf{next}\, !\, P, d\rangle}$$

$$\text{STAR} \quad \frac{n \geq 0}{\langle \star\, P, d\rangle \longrightarrow \langle \mathbf{next}^{\,n} P, d\rangle}$$

$$\text{STR} \quad \frac{\gamma_1 \equiv \gamma_1' \longrightarrow \gamma_2' \equiv \gamma_2}{\gamma_1 \longrightarrow \gamma_2}$$

$$\text{OBS} \quad \frac{\langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow \quad R \equiv F(Q)}{P \xRightarrow{(c,d)} R}$$

TABLE 3.1: Rules for the internal reduction $\longrightarrow$ (upper part) and the observable reduction $\Longrightarrow$ (lower part). Notation $\gamma \not\longrightarrow$ in Rule OBS means that there is no γ' such that $\gamma \longrightarrow \gamma'$. Function F is given in Definition 3.3.

4.1 Interpreting Process Runs.

Let us consider the 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 the environment. At the time unit i, the environment provides a *stimulus* c_i and P_i produces c_i' as *response*. As observers, we can see that on input α the process P responds with α'. We then regard (α, α') as a *reactive observation* of P. If $\alpha = c_1.c_2.c_3\ldots$ and $\alpha' = c_1'.c_2'.c_3'\ldots$, we represent the above interaction as $P \xRightarrow{(\alpha,\alpha')}{}^{\omega}$. Given P we shall refer to the set of all its reactive observations as the *input-output behavior* of P.

Alternatively, if $\alpha = \mathtt{true}^\omega$, we can interpret the run as an interaction among the parallel components in P without the influence of an external environment (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. Thus, as observers what we see is that on the empty input, P produces α in its own right. We shall refer to the set of all timed observations of a process P as the *(default) output* behavior of P.

Another observation we can make of a process is its set of *quiescent input sequences*. Intuitively, those are sequences on input of which P can run without adding any information, wherefore what we observe is that the input and the output coincide. More precisely, the quiescent sequences of P are those sequences α s.t. $P \xrightarrow{(\alpha,\alpha)}{}^{\omega}$.

It turns out that the set quiescent sequences of a process P is equivalent to an observation which is to the opposite extreme of the output behavior observation of P. Namely, the observation of all infinite sequences that P can possibly output under the influence of *arbitrary* environments. Proposition 4.1 below states this equivalence. Consequently, following de Boer *et al.* [1997] in untimed ccp, we shall refer to the set of quiescent sequences of P as the *strongest postcondition* of P (wrt $\mathcal{C}^\omega$), written $sp(P)$. We shall see later that, as in Dijkstra's strongest postcondition approach, proving whether P satisfies a given (temporal) property A, in the presence of any environment, reduces to proving whether $sp(P)$ is included in the set of sequences satisfying A.

The definition below summarizes the various notions of observable behavior above mentioned.

Definition 4.1. (Observables) *The behavioral observations that can be made of a process are:*

(1) The input-output (or stimulus-response) behavior *of P*

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

(2) The (default) output behavior *of P*

$$o(P) = \{\alpha' \mid P \xrightarrow{(\mathtt{true}^\omega,\alpha')}{}^{\omega}\}.$$

(3) The strongest postcondition (or quiescent) *behavior of P*

$$sp(P) = \{\alpha \mid P \xrightarrow{(\alpha,\alpha)}{}^{\omega} \textit{ for some } \alpha\}.$$

The following two sections are devoted to develop a denotational semantics and a temporal logic for the strongest postcondition behavior of P. The input-output and output behavioral observations are studied in detail in [Nielsen and Valencia 2002].

We conclude this section by showing the equivalence above claimed between the set of quiescent sequences of P and the set of all sequences P can possibly output under the presence of arbitrary environments.

PROPOSITION 4.1. (SP AND QUIESCENT BEHAVIOR) *For every ntcc process P and sequence $\alpha \in \mathcal{C}^\omega$, $\alpha \in sp(P)$ iff there exists α' such that $(\alpha', \alpha) \in io(P)$.*

PROOF. The only-if part is trivial: just choose α' as α. Concerning the if part, let $\alpha = c_1.c_2 \ldots c_n \ldots$, $\alpha' = c'_1.c'_2 \ldots c'_n \ldots$. We will prove that from

$$P = P_1 \xRightarrow{(c'_1,c_1)} P_2 \xRightarrow{(c'_2,c_2)} \ldots P_n \xRightarrow{(c'_n,c_n)} \ldots$$

we can derive

$$P = P_1 \xRightarrow{(c_1,c_1)} P_2 \xRightarrow{(c_2,c_2)} \ldots P_n \xRightarrow{(c_n,c_n)} \ldots$$

In order to prove the above, let Q and R be generic processes, and d and e be generic constraints. We will show that $Q \xRightarrow{(d,e)} R$ implies $Q \xRightarrow{(e,e)} R$. An analogous property holds also for standard ccp [de Boer *et al.* 1997].

Note that the observable transition $Q \xRightarrow{(d,e)} R$ corresponds to a sequence of internal transitions

$$\langle Q, d\rangle = \langle Q_1, d_1\rangle \longrightarrow \langle Q_2, d_2\rangle \longrightarrow \ldots \longrightarrow \langle Q_n, d_n\rangle = \langle S, e\rangle \not\longrightarrow$$

with $F(S) = R$. We will show that from the above sequence we can derive

$$\langle Q, e\rangle = \langle Q_1, e\rangle \longrightarrow \langle Q_2, e\rangle \longrightarrow \ldots \longrightarrow \langle Q_n, e\rangle = \langle S, e\rangle \not\longrightarrow$$

Observe that:

(1) During the computation the store increases, i.e. for every P, P', c and c', if $\langle P, c\rangle \longrightarrow \langle P', c'\rangle$ then $c' \vdash c$. This property can be easily proved by induction on the definition of $\longrightarrow$ (Table 3.1). Therefore, $e \vdash d_i$ holds for every $i = 1, 2, \ldots, n$.

(2) For every P, P', c, c' and c'', if $\langle P, c\rangle \longrightarrow \langle P', c'\rangle$ then $\langle P, c \wedge c''\rangle \longrightarrow \langle P', c' \wedge c''\rangle$. Again, the property can be easily proved by induction on the definition of $\longrightarrow$. Hence we have

$$\langle Q_1, d_1 \wedge e\rangle \longrightarrow \langle Q_2, d_2 \wedge e\rangle \longrightarrow \ldots \longrightarrow \langle Q_n, d_n \wedge e\rangle = \langle S, e\rangle \not\longrightarrow$$

(3) Finally from (1) we derive that $d_i \wedge e = e$ for every $i = 1, 2, \ldots, n$, which concludes the proof. $\square$

5. Denotational Semantics

In this section we give a denotational characterization of the strongest postcondition observables of ntcc following ideas developed in [de Boer *et al.* 1997] and [Saraswat *et al.* 1994] for the ccp and tcc case, respectively.

Nevertheless, the combination of nondeterminism and hiding presents a technical problem to deal with: The observables for the hiding operator cannot be specified compositionally (see [de Boer *et al.* 1997]). Another technical problem arises from the combination between hiding and the "unless" operator: the "unless" operator, unlike all other operators of the calculus, is not monotonic. Therefore, our goal is to identify crucial conditions under which the semantics is complete wrt our observables. We shall see that indeed a *significant* fragment of the calculus satisfies such conditions.

The denotational semantics is defined as a function $[\![\cdot]\!]$ which associates to each process a set of infinite constraint sequences, namely $[\![\cdot]\!] : Proc \rightarrow \mathcal{P}(\mathcal{C}^\omega)$. The definition of this function is given in Table 5.2. We use $\exists_x \alpha$ to represent the sequence obtained by applying $\exists_x$ to each constraint in α. Notation $\alpha(i)$ denotes the i-th element in α.

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 5.1. (Fixed Points) The ! and the $\star$ operators are dual. In fact, we could define their denotational semantics as follows:

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

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

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

Theorem 5.1. (Soundness of the Denotation) *For every ntcc process P, $sp(P) \subseteq [\![P]\!]$.*

Proof. Assume $P \xrightarrow{(\alpha,\alpha)}{}^{\omega}$. We proceed by induction on the structure of P.

$P = \mathbf{tell}(c)$**.** Let $\alpha = d.\alpha'$. We must have

$$\mathbf{tell}(c) \xrightarrow{(d,d)} Q \xrightarrow{(\alpha',\alpha')}{}^{\omega}$$

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	$[\![\mathbf{local}\ x\ \mathbf{in}\ P]\!] \;=\; \{\alpha \mid \text{there exists } \alpha' \in [\![P]\!] \text{ s.t. } \exists_x \alpha = \exists_x \alpha'\}$	
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	$[\![\star\, P]\!] \;=\; \{\beta.\alpha \mid \beta \in \mathcal{C}^*, \alpha \in [\![P]\!]\}$	

Table 5.2: Denotational semantics of ntcc

for some Q. But this is possible only if $d \vdash c$. Hence by definition of $[\![\mathbf{tell}(c)]\!]$ we conclude $\alpha \in [\![\mathbf{tell}(c)]\!]$.

$P = \sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$**.** Let $\alpha = d.\alpha'$. We distinguish two cases:

(1) There exists $i \in I$ such that $d \vdash c_i$. Then for some R and Q, $\langle P, d\rangle \longrightarrow \langle P', d\rangle$, $\langle P', d\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow$, and $F(Q) \xRightarrow{(\alpha',\alpha')}{}^{\omega}$. One can verify that $Q \equiv P_i$, hence $P_i \xRightarrow{(d.\alpha',d.\alpha')}{}^{\omega}$. By inductive hypothesis, $d.\alpha' \in [\![P_i]\!]$ and therefore $d.\alpha' \in [\![\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i]\!]$.

(2) For all $i \in I$, $d \not\vdash c_i$. Immediate.

$P = Q \parallel R$**.** Let $\alpha = c_1.c_2 \ldots c_n \ldots$. One can then verify that there is a derivation of the form

$$Q \parallel R = Q_1 \parallel R_1 \xRightarrow{(c_1,c_1)} Q_2 \parallel R_2 \xRightarrow{(c_2,c_2)} \ldots Q_n \parallel R_n \xRightarrow{(c_n,c_n)} \ldots$$

Such that each Q_{i+1} is an evolution of Q_i and each R_{i+1} is an evolution of R_i $(i > 1)$. From the operational semantics of the parallel operator we derive

$$Q = Q_1 \xRightarrow{(c_1,c_1)} Q_2 \xRightarrow{(c_2,c_2)} \dots Q_n \xRightarrow{(c_n,c_n)} \dots$$

and

$$R = R_1 \xRightarrow{(c_1,c_1)} R_2 \xRightarrow{(c_2,c_2)} \dots R_n \xRightarrow{(c_n,c_n)} \dots$$

Therefore, $Q \xRightarrow{(\alpha,\alpha)}{}^{\omega}$ and $R \xRightarrow{(\alpha,\alpha)}{}^{\omega}$. By inductive hypothesis, we have $\alpha \in [\![Q]\!]$ and $\alpha \in [\![R]\!]$, from which we conclude $\alpha \in [\![Q \parallel R]\!]$.

$P =$ **local** x **in** P'**.** Let $\alpha = c_1.c_2 \dots c_n \dots$. We can verify that there must be a derivation of the form

$$\begin{array}{lcl} \textbf{local } x \textbf{ in } P' & = & \textbf{local } x \textbf{ in } P'_1 \xRightarrow{(c_1,c_1)} \\ & & \textbf{local } x \textbf{ in } P'_2 \xRightarrow{(c_2,c_2)} \\ & & \dots \\ & & \textbf{local } x \textbf{ in } P'_n \xRightarrow{(c_n,c_n)} \\ & & \dots \end{array}$$

From the LOC rule for $\longrightarrow$ and induction on the length of each observable transition, we derive

$$P' = P'_1 \xRightarrow{(\exists_x c_1, c'_1)} P'_2 \xRightarrow{(\exists_x c_2, c'_2)} \dots P'_n \xRightarrow{(\exists_x c_n, c'_n)} \dots$$

where for every $i \geq 1$ there exists a d_i such that $c'_i = (\exists_x c_i) \wedge d_i$ and $c_i = c_i \wedge \exists_x d_i$. By Proposition 4.1 we obtain

$$P' = P'_1 \xRightarrow{(c'_1,c'_1)} P'_2 \xRightarrow{(c'_2,c'_2)} \dots P'_n \xRightarrow{(c'_n,c'_n)} \dots$$

and therefore, by the inductive hypothesis, we have $\alpha' = c'_1.c'_2 \dots c'_n \dots$ in $[\![P']\!]$. Finally, observe that $\exists_x \alpha = \exists_x \alpha'$, since for each $i \geq 1$ we have $\exists_x c_i = \exists_x(c_i \wedge \exists_x d_i) = (\exists_x c_i) \wedge (\exists_x d_i) = \exists_x((\exists_x c_i) \wedge d_i) = \exists_x c'_i$.

$P =$ **next** P'**.** Let $\alpha = d.\alpha'$. Then we must have

$$\textbf{next } P' \xRightarrow{(d,d)} Q \xRightarrow{(\alpha',\alpha')}{}^{\omega}$$

for some Q. We must have $Q \equiv F(\textbf{next } P') = P'$. By inductive hypothesis, we then derive $\alpha' \in [\![P']\!]$ and therefore $d.\alpha' \in [\![\textbf{next } P']\!]$.

$P =$ **unless** c **next** P'**.** Let $\alpha = d.\alpha'$. We distinguish two cases:

(1) $d \vdash c$. Immediate.

(2) $d \not\vdash c$. This case is similar to the case $P = \mathbf{next}\, P'$.

$P = !\,Q$. Let $\alpha = c_1.c_2.c_3 \ldots c_n \ldots$. We can verify that if $\langle !\,Q, c_1\rangle \longrightarrow \langle R, c_1\rangle$, then $R \equiv Q \parallel \mathbf{next}\, Q$. We then must have $\langle Q \parallel \mathbf{next}\,!\,Q, c_1\rangle \longrightarrow^* \langle Q' \parallel \mathbf{next}\,!\,Q, c_1\rangle \not\longrightarrow$. Since $F(Q' \parallel \mathbf{next}\,!\,Q) = F(Q') \parallel !\,Q$, by repeating this reasoning and that of the parallel case, we can obtain for $Q = Q_{1,1}$ a derivation of the form:

$$\begin{array}{lll}
!\,Q_{1,1} & \xRightarrow{(c_1,c_1)} & Q_{1,2} \parallel !\,Q_{1,1} \\
& \xRightarrow{(c_2,c_2)} & Q_{1,3} \parallel Q_{2,2} \parallel !\,Q_{1,1} \\
& \xRightarrow{(c_3,c_3)} & Q_{1,4} \parallel Q_{2,3} \parallel Q_{3,2} \parallel !\,Q_{1,1} \\
& \ldots & \\
& \xRightarrow{(c_{n-1},c_{n-1})} & Q_{1,n} \parallel Q_{2,n-1} \parallel Q_{3,n-2} \parallel \ldots \parallel Q_{n-1,2} \parallel !\,Q_{1,1} \\
& \xRightarrow{(c_n,c_n)} & \ldots
\end{array}$$

where each parallel component contributes in the following way:

$$\begin{array}{l}
Q_{1,1} \xRightarrow{(c_1,c_1)} Q_{1,2} \xRightarrow{(c_2,c_2)} Q_{1,3} \ldots \xRightarrow{(c_{n-1},c_{n-1})} Q_{1,n} \xRightarrow{(c_n,c_n)} \ldots \\
Q_{1,1} \xRightarrow{(c_2,c_2)} Q_{2,2} \xRightarrow{(c_3,c_3)} Q_{2,3} \ldots \xRightarrow{(c_{n-1},c_{n-1})} Q_{2,n-1} \xRightarrow{(c_n,c_n)} \ldots \\
Q_{1,1} \xRightarrow{(c_3,c_3)} Q_{3,2} \xRightarrow{(c_4,c_4)} Q_{3,3} \ldots \xRightarrow{(c_{n-1},c_{n-1})} Q_{3,n-2} \xRightarrow{(c_n,c_n)} \ldots \\
\ldots
\end{array}$$

By the inductive hypothesis we derive

$$\begin{array}{r}
c_1.c_2.c_3 \ldots c_n \ldots \in [\![Q]\!] \\
c_2.c_3 \ldots c_n \ldots \in [\![Q]\!] \\
c_3 \ldots c_n \ldots \in [\![Q]\!] \\
\ldots
\end{array}$$

By definition of $[\![!\,Q]\!]$ we conclude $c_1.c_2.c_3 \ldots c_n \ldots \in [\![!\,Q]\!]$.

$P = \star Q$. Let $\alpha = c_1.c_2.c_3 \ldots c_n \ldots$. If $\langle \star Q, c_1\rangle \longrightarrow \langle R, c_1\rangle$ then $R \equiv \mathbf{next}^{\,k} Q$ for some $k \geq 0$. We distinguish two cases.

$k = 0$. In this case we have

$$\langle \star Q, c_1\rangle \longrightarrow \langle Q, c_1\rangle \longrightarrow^* \langle Q', c_1\rangle \not\longrightarrow$$

and $F(Q') \xRightarrow{(\alpha',\alpha')}{}^{\omega}$, where $\alpha' = c_2.c_3 \ldots c_n \ldots$. Hence $Q \xRightarrow{(\alpha,\alpha)}{}^{\omega}$. By inductive hypothesis we derive $\alpha \in [\![Q]\!]$ and therefore, by definition of $[\![\star Q]\!]$, $\alpha \in [\![\star Q]\!]$.

$k \geq 1$**.** Since there are no internal transitions from $\langle \mathbf{next}^k Q, c_1 \rangle$ for $k \geq 1$, and $F(\mathbf{next}^i Q) = \mathbf{next}^{i-1} Q$ for every $i \geq 1$, we must have that

$$\star Q \xRightarrow{(c_1,c_1)} \mathbf{next}^{k-1} Q \xRightarrow{(c_2,c_2)} \ldots \mathbf{next}\, Q \xRightarrow{(c_k,c_k)} Q$$

and

$$Q = Q_1 \xRightarrow{(c_{k+1},c_{k+1})} Q_2 \xRightarrow{(c_{k+2},c_{k+2})} \ldots Q_{n-k} \xRightarrow{(c_n,c_n)} \ldots$$

Hence we derive, by inductive hypothesis, that the sequence $c_{k+1}.c_{k+2}.c_{k+3} \ldots c_n \ldots$ is in $[\![Q]\!]$. By definition of $[\![\star Q]\!]$ we conclude $\alpha \in [\![\star Q]\!]$. □

5.1 Completeness

We now explore the reverse of Theorem 5.1, namely completeness, which does not hold in general. The essential reason is the combination of the **local** operator, which decreases the store by hiding away information, and constructs whose input-output relation is not *conversely monotonic* in the sense described below. Let us first define the following relation on sequences. Recall $\alpha(i)$ denotes the i-th element in α.

Definition 5.1. (Sequence Order) *The (partial) ordering $\leq$ on $\mathcal{C}^\omega$ is defined by $\alpha \leq \alpha'$ iff for all $i \geq 1, \alpha'(i) \vdash \alpha(i)$ holds.*

By a conversely monotonic relation R here we mean that if $(\alpha_1, \alpha_2) \in R$ and $\alpha'_1 \leq \alpha_1$ then there exists α'_2 such that $(\alpha'_1, \alpha'_2) \in R$ and $\alpha'_2 \leq \alpha_2$. We shall call a relation R *monotonic* if whenever $(\alpha_1, \alpha_2) \in R$ and $\alpha_1 \leq \alpha'_1$ then there exists α'_2 such that $(\alpha'_1, \alpha'_2) \in R$ and $\alpha_2 \leq \alpha'_2$.

One example of a construct whose input-output relations $io(.)$ are not necessary conversely monotonic is the choice $\sum_{i \in I}$ **when** c_i **do** P_i when two of the guards (the c_i's) are *compatible* but different. Two constraints c and d are compatible if there exists $e \neq \mathtt{false}$ such that $e \vdash c$ and $e \vdash d$. In literature, non-compatible constraints are also called *mutually exclusive.*

Example 5.1. (Conversely Non-monotonic Summation) Let us consider the following processes

$$P \quad = \quad (\mathbf{when}\ (x = a)\ \mathbf{do}\ \mathbf{tell}(\mathtt{true}))\ +\ (\mathbf{when}\ \mathtt{true}\ \mathbf{do}\ \mathbf{tell}(y = b))$$

Notice that $io(P)$ is not conversely monotonic since process P on input $\alpha_1 = (x = a).\mathtt{true}^\omega$ it can output $\alpha_2 = \alpha_1$ while on input $\alpha'_1 = \mathtt{true}^\omega$ it can only output $\alpha'_2 = (y = b).\mathtt{true}^\omega$

By using the local operator, we can now construct a counterexample to completeness. Consider the process

$$Q \quad = \quad \mathbf{local}\ x\ \mathbf{in}\ P$$

We have $\alpha_1 \in [\![P]\!]$ and consequently, since $\exists_x \alpha'_1 = \exists_x \alpha_1$, $\alpha'_1 \in [\![Q]\!]$. However $\alpha'_1 \notin sp(Q)$.

The above example corresponds to Example 4.2 in [de Boer *et al.* 1997], where an analogous fact is proved for ccp. In [de Boer *et al.* 1997] an even stronger negative result is proved: There exist no denotational semantics $[\![\cdot]\!]$ for ccp such that $[\![P]\!]$ is equal to the strongest postcondition of P for every P. Following [de Boer *et al.* 1997], we could prove an analogous result for *ntcc*.

The choice is the only ccp construct that in general does not satisfy converse monotonicity. In ntcc, however, there is also another operator which does not satisfy converse monotonicity: the **unless** construct.

EXAMPLE 5.2. (CONVERSELY NON-MONOTONIC UNLESS) Let us consider the following "unless" processes

$$\begin{array}{rcl} P & = & \textbf{unless } (x = a) \textbf{ next tell}(y = b) \\ Q & = & \textbf{local } x \textbf{ in } P \end{array}$$

Let α_1 and α_1' be like in Example 5.1. Note that P on input α_1' gives only the output $\mathtt{true}.(y = b).\mathtt{true}^\omega$, and on input α_1 gives only the output α_1. Thus, $io(P)$ is not conversely monotonic. Notice that $io(P)$ is not monotonic either, and that like in Example 5.1, $\alpha_1' \in [\![Q]\!]$ while $\alpha_1' \notin sp(Q)$.

We consider now the conditions under which completeness does hold. Intuitively, we need to make sure that we use locality only in combination with conversely monotonic constructs. This justifies the following definition.

DEFINITION 5.2. (LOCALLY INDEPENDENT PROCESSES) *A ntcc process P is local independent iff both the following conditions are satisfied:*

*(1) For every construct of the form $\sum_{i\in I}$ **when** c_i **do** P_i occurring in P, either*
   - *I is a singleton, or*
   - *the c_i's contain only free variables, i.e. their variables are not in the scope of any **local** operator in P.*

*(2) For every construct of the form **unless** c **next** P occurring in P, c contains only free variables.*

The locally independent processes represent a significant fragment of the calculus. In fact every application examples in this paper but Example 7.5.2 belong to this fragment. Moreover, note that the local-independence condition for the choice operator strictly subsumes the so-called *restricted choice* condition considered by [Falaschi *et al.* 1997]. Restricted choice means that in every choice the guards are either pairwise mutually exclusive or equal.

In fact, it is possible to show that any summation with equal guards (also called *blind choice*) $\sum_{i\in I}$ **when** c **do** P_i is equivalent (i.e., input-output congruent) to

$$\textbf{when } c \textbf{ do } \sum_{i\in I} \textbf{when } \mathtt{true} \textbf{ do } P_i.$$

In untimed ccp, under the mutual exclusion condition the summation $\sum_{i\in I}$ **when** c_i **do** P_i is equivalent to the product

$$\prod_{i\in I} \textbf{when } c_i \textbf{ do } P_i.$$

In ntcc, however, this is not case. The input `false` certainly activates all the P_i's in the product but it only activates one of them in the summation. Notice that since the c_i's are mutually exclusive, `false` is the only constraint that can activate more than one of the P_i's in the product. In the untimed case this mismatch is not a problem since the output will be `false` anyway. In ntcc the output will be `false` in the current time interval but not necessarily in the future ones. Therefore, the activity triggered by one of the P_i's in the next time interval could be quite different from the one triggered by all the P_i's in the next time interval. We can solve this problem by using the "unless" operator to preclude the future activity that the P_i's in the product can trigger on input `false`. More precisely, let $\bar{P}$ the process that results from replacing each top level occurrence of **next** Q with **unless** `false` **next** Q. Then the summation with mutually exclusive guards is equivalent to

$$\prod_{i\in I} \textbf{when } c_i \textbf{ do } \bar{P}_i \parallel \sum_{i\in I} \textbf{when } \texttt{false} \textbf{ do } P_i.$$

The summation on the right chooses one of the P_i if the input is `false`.

On the other hand, local independence is strictly more general than restricted choice, in fact ccp programs which contain only restricted choice are *confluent* , in the sense that the outcome of a process does not depend upon the scheduling strategy on its parallel components [Falaschi *et al.* 1997]. Local independence, on the contrary, does not imply confluence, as shown in the following example.

EXAMPLE 5.3. (NON-CONFLUENT LOCAL INDEPENDENCE) Let P and Q be the locally independent processes defined as follows:

$$\begin{array}{lcl} P & = & (\textbf{when } (x=a) \textbf{ do tell}(\texttt{true})) \;+\; (\textbf{when } \texttt{true} \textbf{ do tell}(y=b)) \\ Q & = & P \parallel \textbf{tell}(x=a) \end{array}$$

Consider the computations of Q in the initial store `true`. If the scheduler selects P first, then we have only a derivation of the form

$$\langle Q, \texttt{true}\rangle \;\longrightarrow^{*}\; \langle \textbf{skip} \parallel \textbf{skip}, y=b \wedge x=a\rangle \;\not\longrightarrow$$

If the scheduler selects **tell**$(x=a)$ first, then we have also the derivation

$$\langle Q, \texttt{true}\rangle \;\longrightarrow^{*}\; \langle \textbf{skip} \parallel \textbf{skip}, x=a\rangle \;\not\longrightarrow$$

Next result shows that local independence is preserved through derivations:

PROPOSITION 5.1. (LOCAL INDEPENDENCE INVARIANCE) *If P is local independent, and $P \xRightarrow{(c,d)} Q$ for some c, d and Q, then also Q is local independent.*

PROOF. Reduction $\longrightarrow$ and function F preserve local independence. □

We prove now that for local independent processes the other direction of Theorem 5.1 holds as well. We first need the following technical definition and lemmata:

DEFINITION 5.3. (RELATION $\preceq$) *The relation $\preceq$ on ntcc processes is the minimal ordering relation that satisfies the following:*

(1) For any ntcc process P, **skip** $\preceq P$.
(2) Let P and Q be ntcc processes, and $E[\ ]$ a ntcc "context", i.e. a ntcc process "with a hole". If $P \preceq Q$, then $E[P] \preceq E[Q]$.
(3) If $P \equiv P' \preceq Q' \equiv Q$ then $P \preceq Q$

Intuitively, $P \preceq Q$ represents the fact that Q contains "at least as much code" as P. From a computational point of view, Q is at least as active as P, and therefore P has at least the resting points of Q, i.e., $sp(Q) \subseteq sp(P)$. The following lemma shows that the "future" operator is monotonic wrt $\preceq$.

LEMMA 5.1. ($\preceq$-MONOTONICITY OF FUTURE) *Let P and Q be ntcc processes. If $P \preceq Q$, then $F(P) \preceq F(Q)$.*

PROOF. Using structural induction on Q, the definition of $\equiv$ and F. □

LEMMA 5.2. *Let P be a sub-term of a local independent process P', and assume that for some constraint c and some processes $P_1 = P, P_2, \ldots P_n$ we have*

$$\langle P_1, c\rangle \longrightarrow \langle P_2, c\rangle \longrightarrow \ldots \langle P_n, c\rangle \not\longrightarrow$$

Let x be a variable which does not occur free in P', let Q be a process such that $Q \preceq P$, and let d_1 be a constraint such that $c \vdash d_1$. Then there exist processes $Q_1 = Q, Q_2, \ldots Q_m$, with $m \leq n$, and constraints $d_2, \ldots d_m$ such that $c \vdash d_i$ for every i with $2 \leq i \leq m$, and

$$\langle Q_1, (\exists_x c) \wedge d_1\rangle \longrightarrow \langle Q_2, (\exists_x c) \wedge d_2\rangle \longrightarrow \ldots \langle Q_m, (\exists_x c) \wedge d_m\rangle \not\longrightarrow$$

with $F(Q_m) \preceq F(P_n)$.

PROOF. By induction on n.

$n = 1$. In this case we have $\langle P_1, c\rangle \not\longrightarrow$. Since $c \vdash (\exists_x c) \wedge d_1$, it is easy to see, by case analysis on P_1, that $\langle Q_1, (\exists_x c) \wedge d_1\rangle \not\longrightarrow$. Furthermore, by Lemma 5.1 we get $F(P_1) \preceq F(Q_1)$.

$n > 1$. If $Q_1 = \mathbf{skip}$ we are done. Otherwise we proceed by case analysis on P_1. We consider only the choice and the **unless** operators; the other cases are easy.

$P_1 = \sum_{j\in J} \mathbf{when}\ e_j\ \mathbf{do}\ R_j$. Since $Q_1 \preceq P_1$, we therefore have $Q_1 \equiv \sum_{j\in J} \mathbf{when}\ e_j\ \mathbf{do}\ S_j$ with $S_j \preceq R_j$ for all $j \in J$. Let $c \vdash e_j$, and $P_2 = R_j$. There are two cases:

$(\exists_x c) \wedge d_1 \vdash e_j$. Let $Q_2 = S_j$ and $d_2 = d_1$. We therefore have $\langle Q_1, (\exists_x c) \wedge d_1\rangle \longrightarrow \langle Q_2, (\exists_x c) \wedge d_2\rangle$. Furthermore, since $Q_2 \preceq P_2$ and $c \vdash d_2$, we can apply the inductive hypothesis to get the conclusion.

$(\exists_x c) \wedge d_2 \not\vdash e_j$. Since $(\exists_x c) \not\vdash e_j$ and $c \vdash e_j$, e_j must contain occurrences of x. Since x is not a free variable, by definition of local independence I must be a singleton. Thus $\langle Q_1, (\exists_x c) \wedge d_1\rangle \not\longrightarrow$. Finally, note that $F(Q_1) = \mathbf{skip}$.

$P_1 = \mathbf{unless}\ e\ \mathbf{next}\ R$. Since $Q_1 \preceq P_1$, $Q_1 \equiv \mathbf{unless}\ e\ \mathbf{next}\ S$ with $S \preceq R$. If $\langle P_1, c\rangle \longrightarrow \langle P_1', c\rangle$ then $P_1' \equiv \mathbf{skip}$ and $c \vdash e$. By definition of local independence, e does not contain occurrences of x, hence $(\exists_x c) \wedge d_1 \vdash e$ and therefore $\langle Q_1, (\exists_x c) \wedge d_1\rangle \longrightarrow \langle \mathbf{skip}, (\exists_x c) \wedge d_1\rangle$. $\square$

LEMMA 5.3. *Let* **local** x **in** P *be a local independent process. Assume that for some constraint* c *and some process* R *we have* $P \stackrel{(c,c)}{\Longrightarrow} R$. *Then for every* $Q \preceq P$ *and for every* d *such that* $\exists_x d = \exists_x c$ *there exists* $S \preceq R$ *such that* **local** x **in** $Q \stackrel{(d,d)}{\Longrightarrow}$ **local** x **in** S.

PROOF. Assume $P \stackrel{(c,c)}{\Longrightarrow} R$, $Q \preceq P$, and $\exists_x d = \exists_x c$. Then there exist some processes $P_1 = P, P_2, \ldots P_n$ with $F(P_n) = R$. such that

$$\langle P_1, c\rangle \longrightarrow \langle P_2, c\rangle \longrightarrow \ldots \langle P_n, c\rangle \not\longrightarrow$$

Let $d_1 = \mathtt{true}$. By Lemma 5.2 there exist some processes $Q_1 = Q, Q_2, \ldots Q_m$, with $m \leq n$, and constraints $d_2, \ldots d_m$ such that $c \vdash d_i$ for every i with $2 \leq i \leq m$, and

$$\langle Q_1, (\exists_x c) \wedge d_1\rangle \longrightarrow \langle Q_2, (\exists_x c) \wedge d_2\rangle \longrightarrow \ldots \langle Q_m, (\exists_x c) \wedge d_m\rangle \not\longrightarrow$$

with $F(Q_m) \preceq F(P_n)$. Since $\exists_x c = \exists_x d$, by repeated application of the LOC rule for $\longrightarrow$ we obtain

$$\begin{array}{rl} \langle \textbf{local}\ (x, d_1)\ \textbf{in}\ Q_1, d \wedge \exists_x d_1 \rangle & \longrightarrow \\ \langle \textbf{local}\ (x, d_2)\ \textbf{in}\ Q_2, d \wedge \exists_x d_2 \rangle & \longrightarrow \\ \ldots & \\ \langle \textbf{local}\ (x, d_m)\ \textbf{in}\ Q_m, d \wedge \exists_x d_m \rangle & \not\longrightarrow \end{array}$$

Observe now that for each i such that $1 \leq i \leq m$ we have $d \vdash \exists_x d = \exists_x c \vdash \exists_x d_i$ and therefore $d \wedge \exists_x d_i = d$. Hence we obtain

$$\textbf{local}\ x\ \textbf{in}\ Q_1 \xRightarrow{(d,d)} F(\textbf{local}\ (x, d_m)\ \textbf{in}\ Q_m) = \textbf{local}\ x\ \textbf{in}\ F(Q_m)$$

Finally, define $S = F(Q_m)$ and note that $S = F(Q_m) \preceq F(P_n) = R$. $\Box$

We are now ready to prove our main result:

Theorem 5.2. (Completeness of the Denotation) *If P is a local independent process, then $[\![P]\!] \subseteq sp(P)$.*

Proof. By induction on the structure of P. The only case for which we need the local independence condition is the **local** operator, and we will discuss this case thoroughly. The cases for the operators **next**, $\|$, ! and $\star$ can be proved easily by reversing the proofs of the corresponding cases in Theorem 5.1 and so we skip them. As for the remaining cases, we give their proofs in full extension, although they are also similar to the reverse of the proofs in Theorem 5.1 except for small technical details.

$P =$ **local** x **in** P'**.** Assume $\alpha \in [\![\textbf{local}\ x\ \textbf{in}\ P']\!]$ and let $\alpha = c_1.c_2 \ldots c_n \ldots$. By definition of $[\![\textbf{local}\ x\ \textbf{in}\ P']\!]$, there exist $\alpha' = c'_1.c'_2 \ldots c'_n \ldots$ such that $\exists_x c'_i = \exists_x c_i$ for every $i \geq 1$ and $\alpha' \in [\![P']\!]$. By inductive hypothesis, we must have a derivation of the form

$$P' = P'_1 \xRightarrow{(c'_1,c'_1)} P'_2 \xRightarrow{(c'_2,c'_2)} \ldots P'_n \xRightarrow{(c'_n,c'_n)} \ldots$$

By Proposition 5.1, each P'_i is local independent. Hence, by repeated application of Lemma 5.3, we derive that there exist $Q_1, Q_2, \ldots Q_n, \ldots$ with $Q_1 = P'$ such that $Q_i \preceq P'_i$ for every $i \geq 2$ and

$$\begin{array}{rcll} \textbf{local}\ x\ \textbf{in}\ P' & = & \textbf{local}\ x\ \textbf{in}\ Q_1 & \xRightarrow{(c_1,c_1)} \\ & & \textbf{local}\ x\ \textbf{in}\ Q_2 & \xRightarrow{(c_2,c_2)} \\ & & \ldots & \\ & & \textbf{local}\ x\ \textbf{in}\ Q_n & \xRightarrow{(c_n,c_n)} \\ & & \ldots & \end{array}$$

Hence we conclude $P = \textbf{local}\ x\ \textbf{in}\ P' \xRightarrow{(\alpha,\alpha)}{}^{\omega}$.

$P = \mathbf{tell}(c)$**.** Assume $\alpha \in [\![\mathbf{tell}(c)]\!]$. Then $\alpha = d.\alpha'$ with $d \vdash c$. Hence we have $\mathbf{tell}(c) \xRightarrow{(d,d)} \mathbf{skip}$. Since $\mathbf{skip} \xRightarrow{(\alpha',\alpha')} {}^{\omega}$, we conclude $\mathbf{tell}(c) \xRightarrow{(d.\alpha',d.\alpha')} {}^{\omega}$.

$P = \sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$**.** Assume $\alpha \in [\![\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i]\!]$. Let $\alpha = d.\alpha'$. We distinguish two cases:

(1) There exists $i \in I$ such that $d \vdash c_i$ and $d.\alpha' \in [\![P_i]\!]$. In this case, we have $\langle P, d\rangle \longrightarrow \langle P_i, d\rangle$ and, by inductive hypothesis, $P_i \xRightarrow{(d.\alpha',d.\alpha')} {}^{\omega}$. Hence we conclude $P \xRightarrow{(d.\alpha',d.\alpha')} {}^{\omega}$.

(2) For all $i \in I$, $d \not\vdash c_i$. Then $\langle P, d\rangle \not\longrightarrow$. Since $F(P) = \mathbf{skip}$, we derive
$$P \xRightarrow{(d,d)} \mathbf{skip} \xRightarrow{(\alpha',\alpha')} {}^{\omega}$$
and therefore $P \xRightarrow{(d.\alpha',d.\alpha')} {}^{\omega}$.

$P = \mathbf{unless}\ c\ \mathbf{next}\ P'$**.** Assume $\alpha \in [\![\mathbf{unless}\ c\ \mathbf{next}\ P']\!]$ and $\alpha = d.\alpha'$. We distinguish two cases:

(1) $d \vdash c$. Then $\langle \mathbf{unless}\ c\ \mathbf{next}\ P', d\rangle \longrightarrow \langle \mathbf{skip}, d\rangle$. Since $\mathbf{skip} \xRightarrow{(d.\alpha',d.\alpha')} {}^{\omega}$, we conclude that $\mathbf{unless}\ c\ \mathbf{next}\ P' \xRightarrow{(d.\alpha',d.\alpha')} {}^{\omega}$.

(2) $d \not\vdash c$ and $\alpha' \in [\![P']\!]$. In this case we should certainly have $\mathbf{unless}\ c\ \mathbf{next}\ P' \xRightarrow{(d,d)} P'$ and, by inductive hypothesis, $P' \xRightarrow{(\alpha',\alpha')} {}^{\omega}$. We then conclude $\mathbf{unless}\ c\ \mathbf{next}\ P' \xRightarrow{(d.\alpha',d.\alpha')} {}^{\omega}$. $\square$

The rest of this section investigates the semantic properties of the class of ntcc processes which are *deterministic* and *monotonic.*

DEFINITION 5.4. (DETERMINISTIC & MONOTONIC PROCESSES) *Let P be a ntcc process.*

- *P is deterministic if it does not contain occurrences of either the choice operator (except when the index set is a singleton) or the $\star$ operator.*
- *P is monotonic if it does not contain the* **unless** *operator.*

Note that tcc processes are deterministic, and that any deterministic process is local independent.

We will show that for processes which are both deterministic and monotonic the semantics allows us to retrieve the input-output relation (which for deterministic processes is a function). We first need to introduce the following definitions.

Definition 5.5. (Laticce Notation)

- *Given a set $S \subseteq \mathcal{C}^\omega$, $min(S)$ denotes the minimal element of S, if it exists.*
- *Given $\alpha \in \mathcal{C}^\omega$, $\uparrow \alpha$ denotes the upward closure of α, namely*

$$\uparrow \alpha = \{\alpha' \mid \alpha \leq \alpha'\}$$

It is possible to show that if $(\mathcal{C}, \vdash)$ is a complete lattice, then also $(\mathcal{C}^\omega, \leq)$ is a complete lattice.

On this ordering we can prove for deterministic monotonic processes a property analogous to what holds for deterministic ccp, namely that the input-output relation is a *closure operator*:

Proposition 5.2. (Deterministic & Monotonic Properties) *If P is a deterministic and monotonic process, then*

(1) $io(P)$ is a function.

(2) $io(P)$ is a closure operator, namely it satisfies the following properties

Extensiveness: *If $(\alpha, \alpha') \in io(P)$ then $\alpha \leq \alpha'$.*
Idempotency: *If $(\alpha, \alpha') \in io(P)$ then $(\alpha', \alpha') \in io(P)$.*
Monotonicity: *If $(\alpha_1, \alpha_2) \in io(P)$ and $\alpha_1 \leq \alpha_1'$, then there exists α_2' such that $(\alpha_1', \alpha_2') \in io(P)$ and $\alpha_2 \leq \alpha_2'$.*

Proof.

(1) Analogous to the standard case for deterministic ccp (see for instance [Saraswat *et al.* 1991]). Note that the interleaving rule for the parallel operator does not introduce any nondeterminism.

(2) **Extensivity:** From Observation 1 in Proposition 4.1. (Note that this property holds for ntcc processes in general.)

Idempotency: From Proposition 4.1. (Also this property holds for ntcc processes in general.)

Monotonicity: It is sufficient to show that for every Q, R, c_1 and c_2, if $\langle Q, c_1\rangle \longrightarrow^* \langle R, c_2\rangle \not\longrightarrow$, then for every $c_1' \vdash c_1$ and Q' such that $Q \preceq Q'$ there exists $c_2' \vdash c_2$ and R' with $F(R) \preceq F(R')$ such that $\langle Q', c_1'\rangle \longrightarrow^* \langle R', c_2'\rangle \not\longrightarrow$. This can be proved by induction on the length of the derivation using the following two properties:

(a) $\longrightarrow$ is monotonic wrt the store, in the sense that, for every Q, R, c_1 and c_2, if $\langle Q, c_1\rangle \longrightarrow \langle R, c_2\rangle$ then for every $c_1' \vdash c_1$ and Q' such that $Q \preceq Q'$ there exists $c_2' \vdash c_2$ and R' with $R \preceq R'$ such that $\langle Q', c_1'\rangle \longrightarrow \langle R', c_2'\rangle$. This property holds for ntcc in general and can be proved easily by induction on the structure of Q'.

(b) For every monotonic Q and c_1, if $\langle Q, c_1\rangle \not\longrightarrow$ then for every $c_1' \vdash c_1$ and Q such that $Q \preceq Q'$ we have either
   - $\langle Q', c_1'\rangle \not\longrightarrow$, or
   - There exists $c_2' \vdash c_2$ and R' with $F(Q) \preceq F(R')$ such that $\langle Q', c_1'\rangle \longrightarrow^* \langle R', c_2'\rangle \not\longrightarrow$ Also this property can be proved easily by induction on the structure of Q'. The restriction to programs which do not contain **unless** constructs is essential here. $\square$

Note that **unless** is the only ntcc operator which introduces non-monotonicity (across time boundaries). All the other ntcc constructs, including those of ccp, are monotonic.

A pleasant property of closure operators is that they can be completely characterized by the set of their fixed points, which in our case are the elements of $sp(P)$. This characterization is expressed by the following corollary, whose proof is standard, given that $io(P)$ is a closure operator.

COROLLARY 5.1. *If P is a deterministic and monotonic process, then $(\alpha, \alpha') \in io(P)$ iff $\alpha' = min(sp(P) \cap \uparrow \alpha)$.*

From this corollary, the soundness theorem, the fact that deterministic processes are local independent, and the completeness theorem, we derive:

THEOREM 5.3. (INPUT-OUTPUT RETRIEVAL FROM SP) *If P is deterministic and monotonic, then $(\alpha, \alpha') \in io(P)$ iff $\alpha' = min([\![P]\!] \cap \uparrow \alpha)$.*

Therefore, the input-output and strongest postcondition observations of deterministic and monotonic processes coincide in the following sense.

COROLLARY 5.2. *Let P and Q be deterministic and monotonic processes. $io(P) = io(Q)$ if and only if $sp(P) = sp(Q)$.*

6. Temporal Logic and Proof System

In this section we define a linear temporal logic for expressing temporal properties of ntcc processes. We shall also give an inference system for proving that a process satisfies a property given in this logic.

6.1 Temporal Logic Syntax

We define a linear temporal logic for expressing properties of ntcc processes.

DEFINITION 6.1. (SYNTAX) *The formulae $A, B, ... \in \mathcal{A}$ are defined by the grammar*

$$A := c \mid A \dot{\Rightarrow} A \mid \dot{\neg} A \mid \dot{\exists}_x A \mid \circ A \mid \Box A \mid \Diamond A$$

Here c denotes an arbitrary constraint which we shall refer to as *atomic proposition*. The intended meaning of the other symbols is the following: $\dot{\Rightarrow}$, $\dot{\neg}$ and $\dot{\exists}$ represent linear-temporal logic implication, negation and existential quantification. These symbols are not to be confused with the symbols $\Rightarrow, \neg$ and $\exists$ in the underlying constraint system. The symbols $\circ$, $\Box$, and $\Diamond$ denote the temporal operators *next*, *always* and *sometime*. We use $A \dot{\vee} B$ as an abbreviation of $\dot{\neg} A \dot{\Rightarrow} B$ and $A \dot{\wedge} B$ as an abbreviation of $\dot{\neg}(\dot{\neg} A \dot{\vee} \dot{\neg} B)$. The temporal true symbol $\dot{\mathtt{true}}$ stands for $\Box\,\mathtt{true}$ and the temporal false symbol $\dot{\mathtt{false}}$ stands for $\dot{\neg}\,\dot{\mathtt{true}}$.

6.2 Temporal Logic Semantics

The standard interpretation structures of linear temporal logic are infinite sequences of states of Manna and Pnueli [1991]. In the case of ntcc, it is natural to replace states by constraints, and consider therefore as interpretations the elements of $\mathcal{C}^\omega$. The semantics of the logic is given in Definition 6.3. Following Manna and Pnueli [1991] we first introduce the notion of *x-variant*. Recall that given a sequence α, $\exists_x \alpha$ denotes the sequence resulting from the application of $\exists_x$ to each element in α.

DEFINITION 6.2. (x-VARIANTS) *Given $c, d \in \mathcal{C}$ we say that d is an x-variant of c iff $\exists_x c = \exists_x d$. Similarly, given $\alpha, \alpha' \in \mathcal{C}^\omega$ we say that α' is an x-variant of α iff $\exists_x \alpha = \exists_x \alpha'$*

Intuitively, α' (d) is an x-variant of α (c) if they are the same except for the information about x.

DEFINITION 6.3. (TEMPORAL SEMANTICS) *We say that $\alpha \in \mathcal{C}^\omega$ is a model of (or that it satisfies) A, notation $\alpha \models A$, if $\langle \alpha, 1\rangle \models A$, where:*

$$
\begin{array}{lll}
\langle \alpha, i\rangle \models c & \textit{iff} & \alpha(i) \vdash c \\
\langle \alpha, i\rangle \models \dot{\neg} A & \textit{iff} & \langle \alpha, i\rangle \not\models A \\
\langle \alpha, i\rangle \models A_1 \dot{\Rightarrow} A_2 & \textit{iff} & \langle \alpha, i\rangle \models A_1 \textit{ implies } \langle \alpha, i\rangle \models A_2 \\
\langle \alpha, i\rangle \models \circ A & \textit{iff} & \langle \alpha, i+1\rangle \models A \\
\langle \alpha, i\rangle \models \Box A & \textit{iff} & \textit{for all } j \geq i \;\; \langle \alpha, j\rangle \models A \\
\langle \alpha, i\rangle \models \Diamond A & \textit{iff} & \textit{there is a } j \geq i \;\; \textit{s.t. } \langle \alpha, j\rangle \models A \\
\langle \alpha, i\rangle \models \dot{\exists}_x A & \textit{iff} & \textit{there is an } x\textit{-variant } \alpha' \textit{ of } \alpha \textit{ s.t. } \langle \alpha', i\rangle \models A.
\end{array}
$$

Notation $\alpha(i)$ denotes the i-th element in α. We define $[\![A]\!]$ to be the collection of all models of A, i.e, $[\![A]\!] = \{\alpha \mid \alpha \models A\}$.

We ought to clarify the role of constraints as atomic propositions in our logic. A temporal formula A expresses properties over sequences of constraints. As an atomic proposition, c expresses a property which is satisfied only by those $e.\alpha'$ such that $e \vdash c$ holds. Therefore, the atomic proposition $\mathtt{false}$ (and consequently $\Box\,\mathtt{false}$) has at least one sequence that satisfies it (e.g. $\mathtt{false}^\omega$). On the contrary the temporal formula $\dot{\mathtt{false}}$ has no models whatsoever. Similarly, the models of the temporal formula $c \dot{\vee} d$ are those $e.\alpha'$ such that either $e \vdash c$ or $e \vdash d$ holds. Therefore, the formula $c \dot{\vee} d$ and the atomic proposition $c \vee d$ may have different models since, in general, one can verify that $e \vdash c \vee d$ may hold while neither $e \vdash c$ nor $e \vdash d$ hold – e.g. consider $e = (x = 1 \vee x = 2)$, $c = (x = 1)$ and $d = (x = 2)$. In contrast, the formula $c \dot{\wedge} d$ and the atomic proposition $c \wedge d$ have the same models since $e \vdash (c \wedge d)$ holds if and only if both $e \vdash c$ and $e \vdash d$ hold.

The above discussion tells us that the operators of the constraint system should not be confused with those of the temporal logic. In particular, the operators $\vee$ and $\dot{\vee}$. This distinction does not make our logic intuitionistic. In fact, classically but not intuitionistically valid statements such as $\dot{\neg} A \dot{\vee} A$ and $\dot{\neg}\dot{\neg} A \dot{\Rightarrow} A$ are also valid in our logic (i.e., all sequences in C^ω are models of these statements).

6.3 Proving properties of ntcc processes

Recall that $sp(P)$ denotes the set of all sequences in $\mathcal{C}^\omega$ that P can possibly output on inputs from arbitrary environments (see Proposition 4.1). In this section we consider the problem of proving assertions of the form "P satisfies A", meaning that every sequence P can possibly output on inputs from arbitrary environments satisfies A.

Definition 6.4. ($P \models A$) *Given a ntcc process P, and a temporal logic formula A, we say that P satisfies A, notation $P \models A$, iff $sp(P) \subseteq [\![A]\!]$.*

Notice for example that $\star\,\mathbf{tell}(c) \models \Diamond c$ since in every infinite sequence output by $\star\,\mathbf{tell}(c)$ on arbitrary inputs there must be an element entailing c. We also have $P = \mathbf{tell}(c) + \mathbf{tell}(d) \models (c \dot{\vee} d)$ as every constraint e output by P entails either c or d. In contrast, $Q = \mathbf{tell}(c \vee d) \not\models (c \dot{\vee} d)$ in general since Q can output a constraint e which certainly entails $c \vee d$ and still entails neither c nor d – e.g. take $e = (x = 1 \vee x = 2)$, $c = (x = 1)$ and $d = (x = 2)$. Notice, however, that $Q \models (c \vee d)$. The reader may now see why we wish to distinguish the temporal formula $c \dot{\vee} d$ from the atomic proposition $c \vee d$.

In order to reason about statements of the form $P \models A$, we propose a proof system for assertions of the form $P \vdash A$. Intuitively, we want $P \vdash A$ to be the "counterpart" of $P \models A$ in the inference system, namely $P \vdash A$ should approximate $P \models A$ as closely as possible (ideally, they should be equivalent).

The system is presented in Table 6.3. Rule P1 gives a proof saying that every output of $\mathbf{tell}(c)$ on inputs of arbitrary environments should definitely satisfy the atomic proposition c, i.e., $\mathbf{tell}(c) \models c$. Rule P2 can be explained as follows. Suppose that given a summation $P = \sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$ we have a proof that each P_i satisfies A_i. Then we certainly have a proof saying that every output of P on arbitrary inputs should satisfy either: (a) some of the guards c_i and their corresponding A_i (i.e., $\dot{\bigvee}_{i \in I}(c_i \dot{\wedge} A_i)$), or (b) none of the guards (i.e., $\dot{\bigwedge}_{i \in I} \dot{\neg}\, c_i$). The other rules can be explained in a similar way.

P1	$\mathbf{tell}(c) \;\vdash\; c$
P2	$\dfrac{\forall i \in I \quad P_i \;\vdash\; A_i}{\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i \;\vdash\; \dot{\bigvee}_{i\in I}(c_i \mathbin{\dot\wedge} A_i) \mathbin{\dot\vee} \dot{\bigwedge}_{i\in I} \dot\neg c_i}$
P3	$\dfrac{P \;\vdash\; A \qquad Q \;\vdash\; B}{P \parallel Q \;\vdash\; A \mathbin{\dot\wedge} B}$
P4	$\dfrac{P \;\vdash\; A}{\mathbf{local}\ x\ \mathbf{in}\ P \;\vdash\; \dot\exists_x A}$
P5	$\dfrac{P \;\vdash\; A}{\mathbf{next}\ P \;\vdash\; \circ A}$
P6	$\dfrac{P \;\vdash\; A}{\mathbf{unless}\ c\ \mathbf{next}\ P \;\vdash\; c \mathbin{\dot\vee} \circ A}$
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 $A \mathbin{\dot\Rightarrow} B$

TABLE 6.3: A proof system for linear-temporal properties of ntcc processes

DEFINITION 6.5. ($P \vdash A$) *We say that $P \vdash A$ iff the assertion $P \vdash A$ has a proof in the system in Table 6.3.*

The rest of the section is devoted to exploring the relation between $P \models A$ and $P \vdash A$. To this purpose, it will be useful to introduce the concept of *strongest temporal formula derivable for a process.*

DEFINITION 6.6. (STRONGEST TEMPORAL FORMULA) *A formula A is the strongest temporal formula derivable for a process P if*

(1) $P \vdash A$ and

(2) for every formula A' such that $P \vdash A'$, we have $A \mathbin{\dot\Rightarrow} A'$.

Note that the strongest temporal formula of a process P, if it exists, is unique modulo logical equivalence. We give now a constructive proof of the existence of such a formula for every process P.

DEFINITION 6.7. ($stf(P)$) *The function stf : Proc $\rightarrow \mathcal{A}$ is defined as follows:*

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

We show now that $stf(P)$ is the formula characterized by Definition 6.6.

PROPOSITION 6.1. *For every process P, $stf(P)$ is the strongest temporal formula derivable for P.*

PROOF.
1. $P \vdash stf(P)$ can proven by structural induction on P.
2. $P \vdash A$ implies $stf(P) \mathbin{\dot\Rightarrow} A$ can be proven by induction on (the depth) of the inference; considering all possible cases for the final step of the inference of $P \vdash A$.$\Box$

Furthermore, we have a complete correspondence between the denotational semantics of P and the models of $stf(P)$:

LEMMA 6.1. *For every process P, $[\![P]\!] = [\![stf(P)]\!]$.*

PROOF. By structural induction on P. $\Box$

The following proposition establishes that in the inference system every processes can be proven to satisfy the "always true" temporal formula, and no process can be proven to satisfy the "always false" temporal formula. It also gives us some derived inference rules which will save us some work when proving properties of our application examples (Section 7.2). It follows from the two previous results and the soundness of the denotational semantics.

PROPOSITION 6.2. *For every process P,*
(1) $P \vdash \dot{\mathtt{true}}$
(2) $P \not\vdash \dot{\mathtt{false}}$
(3) $\dfrac{P \vdash A}{P \parallel Q \vdash A}$
(4) $\dfrac{P \vdash A \quad P \vdash B}{P \vdash A \mathbin{\dot\wedge} B}$

Proof. (1) Trivially, $stf(A) \mathrel{\dot{\Rightarrow}} \dot{\texttt{true}}$. Hence,

$$\dfrac{\dfrac{}{P \vdash stf(P)}\ \text{Proposition 6.1}}{P \vdash \dot{\texttt{true}}}\ P9$$

(2) Suppose that for some P, $P \vdash \dot{\texttt{false}}$. We have $[\![\dot{\texttt{false}}]\!] = \emptyset$. From the operational semantics it is easy to see that every process P, on any input it produces some output. Hence $[\![\dot{\texttt{false}}]\!] \subset sp(P)$. From the soundness of the denotational semantics (Theorem 5.1) and Lemma 6.1 we have $sp(P) \subseteq [\![P]\!] = [\![stf(P)]\!]$. From Proposition 6.1 $stf(P)$ is the strongest formula derivable for P, thus by definition of $\dot{\Rightarrow}$, $[\![stf(P)]\!] \subseteq [\![\dot{\texttt{false}}]\!]$, a contradiction.

(3) We have the following derivation

$$\dfrac{\dfrac{P \vdash A \qquad \dfrac{}{Q \vdash \dot{\texttt{true}}}\ 1}{P \parallel Q \vdash A \mathrel{\dot{\wedge}} \dot{\texttt{true}}}\ \text{P3}}{P \parallel Q \vdash A}\ \text{P9}$$

(4) From Proposition 6.1 it follows that $stf(P) \mathrel{\dot{\Rightarrow}} A \mathrel{\dot{\wedge}} B$. Therefore,

$$\dfrac{\dfrac{}{P \vdash stf(P)}\ \text{Proposition 6.1}}{P \vdash\ A \mathrel{\dot{\wedge}} B}\ \text{P9} \quad \square$$

Finally, we have the following result, which states the correspondence between $P \vdash A$ and the semantics counterparts of P and A:

Theorem 6.1. *For every ntcc process P and every formula A, $P \vdash A$ iff $[\![P]\!] \subseteq [\![A]\!]$.*

Proof.

$$\begin{array}{l}
P \vdash A \\
\quad \text{iff} \quad \text{(by Proposition 6.1)} \\
stf(P) \mathrel{\dot{\Rightarrow}} A \\
\quad \text{iff} \quad \text{(by definition of } \dot{\Rightarrow}) \\
[\![stf(P)]\!] \subseteq [\![A]\!] \\
\quad \text{iff} \quad \text{(by Lemma 6.1)} \\
[\![P]\!] \subseteq [\![A]\!] \ \square
\end{array}$$

From Theorems 6.1, 5.1 and 5.2 we immediately derive the following relations between $\models$ and $\vdash$:

Corollary 6.1. (Soundness of the Logic)
For every ntcc process P and every formula A, if $P \vdash A$ then $P \models A$.

Corollary 6.2. (Completeness of the Logic) *For every local independent process P and every formula A, if $P \models A$ then $P \vdash A$.*

The reason why this theorem is called "relative completeness" is because we need to determine the validity of the temporal implication in consequence rule P9. The validity problem in the linear-time temporal logic of Manna and Pnueli [1991] is known to be decidable for the quantifier-free fragment as well as for some other interesting first-order fragments (see [Hodkinson *et al.* 2000]). We therefore have reasons to believe that the validity problem is decidable also for interesting fragments of our logic. This topic is currently being investigated by the authors.

Note that from the results of this section it follows that in order to prove $P \vdash A$ it would be sufficient to prove $\mathit{stf}(P) \Rightarrow A$. Proving such implication, however, may not be simple. As pointed-out in [Manna and Pnueli 1991], it has the disadvantage that, even for a simple A such as $\Diamond(x > 1)$ the formula $\mathit{stf}(P) \Rightarrow A$ carries the full complexity of $\mathit{stf}(P)$. The inference 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.

7. Applications

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

DEFINITION 7.1. (A FD CONSTRAINT SYSTEM) *Define* $FD[max]$*, where* $max > 1$*, as the constraint system whose signature* Σ *includes symbols in* $\{0, 1, ..., max-1, \mathtt{succ}, \mathtt{prd}, +, \times, \mathtt{mod}, =\}$ *and 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 $\mathcal{D}$ as the set $\{0, 1, ..., max-1\}$ and use v and w to range over its elements. We shall often just write v (w) instead of the expression $v \in \mathcal{D}$ $(w \in \mathcal{D})$ if $\mathcal{D}$ is being used as an indexing set.

7.1 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 is restricted to call q at most once and such a call must be within the scope of a "**next**". 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. Furthermore, we consider call-by-value in the sense that the intended behavior of a call $q(t)$, where t is a term fixed to a value v (i.e., the current store entails $t = v$), is that of $P_q[v/x]$, where $[v/x]$ is the operation of (syntactical) replacement of every occurrence of x by v.

As in the π-calculus we do not want, when unfolding recursive calls of q, the free variables of P_q (see Definition 3.1) to get captured in the lexical-scope of a bound-variable in P_q. The following example should make this matter clearer.

Example 7.1.

$$q(x) \stackrel{\text{def}}{=} (\textbf{when } y = 1 \textbf{ do tell}(c)\ ) \ \parallel\ \textbf{local } y \textbf{ in next}(q(x) \parallel \textbf{tell}(y = 1))$$

Notice that if we unfold the recursive call of $q(x)$ within the **next**, the variable y in the guard $y = 1$ gets captured in the lexical-scope of the bound-variable y. It easy to see that $q(0)$ would always tell c after the next time unit.

We shall avoid this kind of capture by requiring $fv(P_q) \subseteq \{x\}$, if a recursive call of q occurs within P_q in the lexical-scope of a bound-variable. For clarity, we shall then sometimes write $q(x)[y_1, \ldots, y_n] \stackrel{\text{def}}{=} P_q$, to explicitly mention that variables $y_1, \ldots, y_n$ may occur free in P_q.

7.1.1 The Encoding.

First we introduce some notation. Given $q(x) \stackrel{\text{def}}{=} P_q$, we will use $q, qarg$ to denote any two variables not in $fv(P_q)$. We use $x \leftarrow t$ to denote the process $\sum_v \textbf{when}\, t = v\, \textbf{do}\, !\, \textbf{tell}(x = v)$. Intuitively, $x \leftarrow t$ denotes the persistent assignment of t's fixed value, say v, to x. Furthermore, let $\ulcorner P \urcorner$ the process obtained by replacing in P any call $q(t)$ with $\textbf{tell}(\mathsf{call}(q)) \parallel \textbf{tell}(qarg = t)$. The idea is that such a replacement will tell that there is a call of q with argument t and excite a new copy of P_q.

The process corresponding to definition of $q(x)$, denoted as $\ulcorner q(x) \stackrel{\text{def}}{=} P_q \urcorner$, is:

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

Thus, 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 $\ulcorner P_q \urcorner$.

Finally, we consider the calls $q(t)$ in other processes. Each such a call is replaced by

$$\textbf{local}\, q\ qarg\, \textbf{in}\, (\ulcorner q(x) \stackrel{\text{def}}{=} P_q \urcorner \parallel \textbf{tell}(\mathsf{call}(q)) \parallel \textbf{tell}(qarg = t)),$$

which we shall denote by $\ulcorner q(t) \urcorner$. The local declarations are needed to avoid interference with other recursive calls.

7.1.2 Parameterless Recursion.

Our encoding generalizes easily to the case of an arbitrary number of parameters. We will be using parameterless recursion as well, thus we shall give the encoding for this particular the case. So, given a definition $q \stackrel{\text{def}}{=} P_q$ we define its encoding as $\ulcorner q \stackrel{\text{def}}{=} P_q \urcorner =!\,(\textbf{when}\ \mathsf{call}(q)\ \textbf{do} \ulcorner P_q \urcorner)$, where in this case $\ulcorner P_q \urcorner$ results from replacing in P_q each call q by $\textbf{tell}(\mathsf{call}(q))$. The calls to definition q in other process should be replaced by the process $\ulcorner q \urcorner = \textbf{local}\, q\, \textbf{in}\, (\ulcorner q \stackrel{\text{def}}{=} P_q \urcorner \parallel \textbf{tell}(\mathsf{call}(q)))$.

We finish this section with a result which gives us a proof principle for proving temporal properties of recursive definitions. The next lemma 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 $t = v$.

LEMMA 7.1. (BASIC TEMPORAL PROPERTY OF RECURSION) *Given a definition $\ulcorner q(x) \stackrel{\text{def}}{=} P_q \urcorner$, suppose that $q, qarg$ do not occur free in B and that $\ulcorner P_q \urcorner \vdash B$. Then for all $v \in \mathcal{D}$, $\ulcorner q(t) \urcorner \vdash t = v \dot{\Rightarrow} B[v/x]$.*

PROOF. From Proposition 6.1 we obtain the the strongest formula F derivable for $\ulcorner q(t) \urcorner$, i.e.,

$$F = \dot{\exists}_{q,qarg}(\mathsf{call}(q) \mathrel{\dot{\wedge}} qarg = t \mathrel{\dot{\wedge}} \Box(\mathsf{call}(q) \Rightarrow \dot{\exists}_x(A \mathrel{\dot{\wedge}} H))$$

where A is the strongest-formula derivable for $\ulcorner P_q \urcorner$ and H is the formula $\dot{\bigvee}_w(qarg = w \mathrel{\dot{\wedge}} \Box x = w) \mathrel{\dot{\vee}} \dot{\bigwedge}_w \dot{\neg}\, x = w$. From Definition 6.6, $A \dot{\Rightarrow} B$. Let G be the formula obtained from replacing A with B in F. Thus, $F \dot{\Rightarrow} G$. From Rule P9 we get $\ulcorner q(t) \urcorner \vdash G$. Now let v an arbitrary value in $\mathcal{D}$. By observing that q does not occur free in B and some manipulations we verify that

$$G \dot{\Rightarrow} G' = \dot{\exists}_{qarg}(qarg = t \mathrel{\dot{\wedge}} \dot{\exists}_x(B \mathrel{\dot{\wedge}} H)).$$

From Rule P9 we get $\ulcorner q(t) \urcorner \vdash G'$. By observing that $qarg$ does not occur free in B, we verify that $G' \dot{\Rightarrow} (t = v \dot{\Rightarrow} B[v/x])$. By applying Rule P9 we get the desired result, i.e., $\ulcorner q(t) \urcorner \vdash t = v \dot{\Rightarrow} B[v/x]$. $\Box$

For parameterless recursion we have the following version of the above lemma.

LEMMA 7.2. (PROPERTY OF PARAMETERLESS RECURSION) *Given a definition $\ulcorner q \stackrel{\text{def}}{=} P_q \urcorner$, suppose that q does not occur free in B and $\ulcorner P_q \urcorner \vdash B$. Then $\ulcorner q \urcorner \vdash B$.*

7.2 Cell Example

Cells provide a basis for the specification and analysis of mutable and persistent data structures. Let us assume that the signature is extended with an unary predicate symbol `change`. A *mutable cell* $x\colon(v)$ can be viewed as a structure x which has a current value v and which can, in the future, be assigned a new value.

$$
\begin{array}{lll}
x\colon (z) & \stackrel{\text{def}}{=} & \mathbf{tell}(x=z) \parallel \mathbf{unless}\ \mathtt{change}(x)\ \mathbf{next}\ x\colon (z) \\
\mathrm{exch}_g[x,y] & \stackrel{\text{def}}{=} & \sum_v \mathbf{when}\ x=v\ \mathbf{do}\,(\ \mathbf{tell}(\mathtt{change}(x)) \parallel \mathbf{tell}(\mathtt{change}(y)) \\
 & & \qquad\qquad \parallel \mathbf{next}(\ \ulcorner x\colon (g(v))\urcorner \parallel \ulcorner y\colon (v)\urcorner)\ \ )
\end{array}
$$

Definition $x : (z)$ represents a cell x whose value is z and it will be the same in the next time interval unless it is to be changed next (i.e., $\mathtt{change}(x)$). Definition $\mathrm{exch}_g[x,y]$ represents an *exchange* operation in the following sense: if v is x's current value then $g(v)$ and v will be the next values of x and y respectively. The following proposition describe the exchange operation in logical terms.

Proposition 7.1. (A Temporal Property of Cells) *For every value* $v \in \mathcal{D}$, $\ulcorner \mathrm{exch}_g[x,y]\urcorner \vdash x = v \mathrel{\dot{\Rightarrow}} \circ(x = g(v) \mathrel{\dot{\wedge}} y = v)$ *.*

Below we describe in detail the proof of the above proposition to illustrate the inference system at work. The abbreviations *Pr.* and *Eq.* in the derivations below stands for "*Proposition*" and "*Equation*", respectively. Recall P1,P2,. . . ,P9 are the labels of the inference rules in Table 6.3.

Consider $x\colon (z) \stackrel{\text{def}}{=} P_{x:} = \mathbf{tell}(x = z) \parallel \mathbf{unless}\ \mathtt{change}(x)\ \mathbf{next}\ x\colon (z)$. Thus, $\ulcorner P_{x:}\urcorner = \mathbf{tell}(x = z) \parallel U$ with $U = \ulcorner\mathbf{unless}\ \mathtt{change}(x)\ \mathbf{next}\ x\colon (z)\urcorner$ – see the "$\ulcorner . \urcorner$" notations in the encoding of recursion and parameterless recursion, Section 7.1.1.

We have the following derivation (proof) of

$$\ulcorner P_{x:}\urcorner \vdash x = z. \tag{7.1}$$

$$
\dfrac{\dfrac{}{\mathbf{tell}(x=z) \vdash x = z}\ \mathrm{P1}}{\ulcorner P_{x:}\urcorner \vdash x = z}\ Pr.6.2(3)
$$

For $y\colon (z) \stackrel{\text{def}}{=} P_{y:} = \mathbf{tell}(y = z) \parallel \mathbf{unless}\ \mathtt{change}(y)\ \mathbf{next}\ y\colon (z)$, we get a similar derivation of

$$\ulcorner P_{y:}\urcorner \vdash y = z. \tag{7.2}$$

By applying Lemma 7.1 to (7.1) and (7.2) we get that for arbitrary g, w

$$\ulcorner x : (g(w))\urcorner \vdash x = g(w) \text{ and } \ulcorner y : (w)\urcorner \vdash y = w. \tag{7.3}$$

Let us consider the definition $\mathrm{exch}_g[x,y] \stackrel{\text{def}}{=} Q = \sum_w \mathbf{when}\ (x = w)\ \mathbf{do}\ Q_w$, where each $Q_w = R \parallel \mathbf{next}(\ulcorner x : (g(w))\urcorner \parallel \ulcorner y : (w)\urcorner)$ with R being the process $\mathbf{tell}(\mathtt{change}(x)) \parallel \mathbf{tell}(\mathtt{change}(y))$.

Since there are no recursive calls $\ulcorner Q_w\urcorner = Q_w$ and $\ulcorner Q\urcorner = Q$. The following is a derivation of

$$\ulcorner Q\urcorner \vdash (x = v \mathrel{\dot{\Rightarrow}} \circ(x = g(v) \mathrel{\dot{\wedge}} y = v)). \tag{7.4}$$

$$
\dfrac{
  \forall w \in \mathcal{D} \qquad
  \dfrac{
    \dfrac{
      \dfrac{
        \dfrac{}{x:(g(w)) \vdash x = g(w)}\,Eq.(7.3)
        \qquad
        \dfrac{}{y:(w) \vdash y = w}\,Eq.(7.3)
      }{
        \ulcorner x:(g(w))\urcorner \parallel \ulcorner y:(w)\urcorner \vdash x = g(w) \mathbin{\dot\wedge} y = w
      }\,\mathrm{P3}
    }{
      \mathbf{next}(\ulcorner x:(g(w))\urcorner \parallel \ulcorner y:(w)\urcorner) \vdash \bigcirc(x = g(w) \mathbin{\dot\wedge} y = w)
    }\,\mathrm{P5}
  }{
    Q_w \vdash \bigcirc(x = g(w) \mathbin{\dot\wedge} y = w)
  }\,Pr.6.2(3)
}{
  \dfrac{
    Q \vdash \dot{\bigvee}_{w\in\mathcal{D}} (x = w \mathbin{\dot\wedge} \bigcirc(x = g(w) \mathbin{\dot\wedge} y = w)) \mathbin{\dot\vee} \dot{\bigwedge}_{w\in\mathcal{D}} \dot\neg\, x = w
  }{
    \dfrac{
      Q \vdash \dot{\bigwedge}_{w\in\mathcal{D}} (x = w \mathbin{\dot\Rightarrow} \bigcirc(x = g(w) \mathbin{\dot\wedge} y = w))
    }{
      Q \vdash (x = v \mathbin{\dot\Rightarrow} \bigcirc(x = g(v) \mathbin{\dot\wedge} y = v))
    }\,\mathrm{P9}
  }\,\mathrm{P9}
}\,\mathrm{P2}
$$

From Equation (7.4) and Lemma 7.2 we obtain

$$\ulcorner \mathrm{exch}_g[x,y]\urcorner \vdash x = v \mathbin{\dot\Rightarrow} \bigcirc(x = g(v) \mathbin{\dot\wedge} y = v)$$

as wanted $\Box$.

Often in operations $\mathrm{exch}_g[x,y]$ we use functions g that always return the same value (i.e. constants). It is convenient to take the liberty of using that value as its symbol. Let us consider the following example

Example 7.2. The execution of $\ulcorner x\colon(3)\urcorner \parallel \ulcorner y\colon(5)\urcorner \parallel \ulcorner \mathrm{exch}_7[x,y]\urcorner$ will give us in the next time unit the cells x and y with values 7 and 3, respectively. In fact, we get the following derivation

$$
\dfrac{
  \dfrac{
    D \qquad
    \dfrac{}{\ulcorner \mathrm{exch}_7[x,y]\urcorner \vdash x = 3 \mathbin{\dot\Rightarrow} \bigcirc(x = 7 \mathbin{\dot\wedge} y = 3)}\,\mathit{Pr.7.1}
  }{
    \ulcorner x:(3)\urcorner \parallel \ulcorner y:(5)\urcorner \parallel \ulcorner \mathrm{exch}_7[x,y]\urcorner \vdash x = 3 \mathbin{\dot\wedge} (x = 3 \mathbin{\dot\Rightarrow} \bigcirc(x = 7 \mathbin{\dot\wedge} y = 3))
  }\,\mathrm{P3}
}{
  \ulcorner x:(3)\urcorner \parallel \ulcorner y:(5)\urcorner \parallel \ulcorner \mathrm{exch}_7[x,y]\urcorner \vdash \bigcirc(x = 7 \mathbin{\dot\wedge} y = 3)
}\,\mathrm{P9}
$$

where D is the derivation

$$
\dfrac{
  \dfrac{}{\ulcorner x:(3)\urcorner \vdash x = 3}\,\mathit{Eq.(7.1)}
}{
  \ulcorner x:(3)\urcorner \parallel \ulcorner y:(5)\urcorner \vdash x = 3
}\,\mathit{Pr.6.2(3)}
$$

Assignments, increasing and decreasing operations are typical cell operations. The assignment of v to a cell x, written $x := v$, can then be encoded as **local** y **in** $\ulcorner \mathrm{exch}_v(x,y)\urcorner$ where the local variable y is used as dummy variable (cell). Similarly, we can encode instructions $x := x+1$ and $x := x-1$ by using dummy variables, and the `succ` and `prd` functions. We shall use these constructs in the next sections.

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

Proposition 7.2. (Another Cell Temporal Property) *For all values* $v \in \mathcal{D}$, $\ulcorner x\colon(v)\urcorner \vdash (A \mathbin{\dot\wedge} \dot\neg\, \mathtt{change}(x)) \mathbin{\dot\Rightarrow} \bigcirc A$.

Proof. By verifying that $\mathit{stf}(\ulcorner x:(v)\urcorner) \mathbin{\dot\Rightarrow} (A \mathbin{\dot\wedge} \dot\neg\, \mathtt{change}(x)) \mathbin{\dot\Rightarrow} \bigcirc A)$. $\Box$

7.3 RCX Robots: The Zigzagging Example

An RCX is a programmable, controller-based LEGO® brick used to create autonomous robotic devices [Lund and Pagliarini 1999]. Zigzagging [Fredslund 2000] 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. We use cells a_1 and a_2 to "look back" one and two time units, respectively. We use three distinct constants $\mathtt{f},\mathtt{r},\mathtt{l} \in \mathcal{D} - \{0\}$ and extend the signature with the predicate symbols forward,right,left.

$$
\begin{array}{lcl}
\mathit{GoF} & \stackrel{\text{def}}{=} & \ulcorner \mathrm{exch}_{\mathtt{f}}[a_1, a_2] \urcorner \parallel \mathbf{tell}(\mathtt{forward}) \\
\mathit{GoR} & \stackrel{\text{def}}{=} & \ulcorner \mathrm{exch}_{\mathtt{r}}[a_1, a_2] \urcorner \parallel \mathbf{tell}(\mathtt{right}) \\
\mathit{GoL} & \stackrel{\text{def}}{=} & \ulcorner \mathrm{exch}_{\mathtt{l}}[a_1, a_2] \urcorner \parallel \mathbf{tell}(\mathtt{left}) \\
\mathit{Zigzag} & \stackrel{\text{def}}{=} & !\,(\quad \mathbf{when}\ (a_1 \neq \mathtt{f})\ \mathbf{do}\ \ulcorner \mathit{GoF} \urcorner \\
 & & \quad + \ \mathbf{when}\ (a_2 \neq \mathtt{r})\ \mathbf{do}\ \ulcorner \mathit{GoR} \urcorner \\
 & & \quad + \ \mathbf{when}\ (a_2 \neq \mathtt{l})\ \mathbf{do}\ \ulcorner \mathit{GoL} \urcorner\ ) \\
 & & \\
\mathit{GoZigzag} & \stackrel{\text{def}}{=} & \ulcorner a_1 \colon (0) \urcorner \parallel \ulcorner a_2 \colon (0) \urcorner \parallel \ulcorner \mathit{Zigzag} \urcorner.
\end{array}
$$

Initially cells a_1 and a_2 contain neither $\mathtt{f}, \mathtt{r}$ nor $\mathtt{l}$. After a choice is made according to (1), (2) and (3), it is recorded in a_1 and the previous one moved to a_2.

One can verify that $\ulcorner \mathit{GoZigzag} \urcorner \models \Box(\Diamond \mathtt{right} \dot{\wedge} \Diamond \mathtt{left})$, thus stating that the robot indeed goes right and left infinitely often. Since $\ulcorner \mathit{GoZigzag} \urcorner$ is a locally independent process, from the completeness of our inference it follows that:

Proposition 7.3. (A Zigzag Temporal Property)

$$\ulcorner \mathit{GoZigzag} \urcorner \vdash \Box(\Diamond \mathtt{right} \dot{\wedge} \Diamond \mathtt{left}).$$

7.4 Multi-Agent Systems: The Pursuit Game Example

The Predator/Prey (or Pursuit) game [Benda *et al.* 1986] has been studied using a wide variety of approaches [Haynes and Sen 1996] and it has many different instantiations that can be used to illustrate different multi-agent scenarios [Stone and Veloso 2000]. As the Zigzagging example, instances of the Predator/Prey game have been modeled using autonomous robots [Nolfi and Floreano 1998]. Here we model a simple instance of this game.

The predators and prey move around in a discrete, grid-like toroidal world with square spaces; they can move off one end of the board and come back on the other end. Predators and prey move simultaneously. They can move vertically and horizontally in any direction. In order to simulate fast but not very precise predators and a slower but more maneuverable prey we assume that predators move two squares in straight line while the prey moves just one.

The goal of the predators is to "capture" the prey. A capture position occurs when the prey moves into a position which is within the three-squares line of a predator current move; i.e. if for some of the predators, the prey current position is either the predator current position, the predator previous position, or the square between these two positions. This simulates the prey deadly moving through the line of attack of a predator.

For simplicity, we assume that initially the predators are in the same row immediately next to each other, while the prey is in front of a predator (i.e, in the same column, above this predator) one square from it. The prey's maneuver to try to escape is to move in an unpredictable zigzagging around the world. The strategy of the predators is to cooperate to catch the prey. Whenever one of the predators is in front of the prey it declares itself as the leader of the attack and the other becomes its support. Therefore depending on the moves of the prey the role of leader can be alternated between the predators. The leader moves towards the prey, i.e. if it sees the prey above it then it moves up, if it sees the prey below it then it moves down, and so on. The support predator moves in the direction the leader moves, thus making sure it is always next to leader.

In order to model this example we extend the signature with the predicates symbols $\mathtt{right}_i, \mathtt{left}_i, \mathtt{up}_i, \mathtt{down}_i$ for $i \in \{0, 1\}$. For simplicity we assume there are only two predators $Pred_0$ and $Pred_1$. We use the cells x_i, y_i and cells x, y for representing the current positions of predator i and the prey, respectively, in a $max \times max$ matrix (we assume that $max = 2^k$ for some $k > 1$) representing the world. We also use the primed version of these cells to keep track of corresponding previous positions and cell l to remember which predator is the current leader. We can now formulate the capture condition. Predator i captures the prey with a horizontal move iff

$$\begin{array}{ll} x'_i = x = x_i \wedge (& (y_i = y'_i - 2 \wedge (y = y'_i \vee y = y'_i - 1 \vee y = y'_i - 2)) \vee \\ & (y_i = y'_i + 2 \wedge (y = y'_i \vee y = y'_i + 1 \vee y = y'_i + 2)) \ ) \end{array}$$

and with a vertical move iff

$$\begin{array}{ll} y'_i = y = y_i \wedge (& (x_i = x'_i - 2 \wedge (x = x'_i \vee x = x'_i - 1 \vee x = x'_i - 2)) \vee \\ & (x_i = x'_i + 2 \wedge (x = x'_i \vee x = x'_i + 1 \vee x = x'_i + 2)) \ ). \end{array}$$

We define $\mathtt{capture}_i$ as the conjunction of the two previous constraints.

The process below models the behavior of the prey. The prey moves as in the Zigzagging example. Furthermore, the values of cells x, y and x', y' are updated according to the zigzag move (e.g., if it goes right the value of x is increased and x' takes x's previous value).

$$\begin{array}{llllll} Prey \stackrel{\text{def}}{=} \ulcorner GoZigzag \urcorner \parallel & !(& \textbf{when} & \mathtt{forward} & \textbf{do} & \ulcorner \mathrm{exch}_{\mathtt{prd}}[y, y'] \urcorner \\ & + & \textbf{when} & \mathtt{right} & \textbf{do} & \ulcorner \mathrm{exch}_{\mathtt{succ}}[x, x'] \urcorner \\ & + & \textbf{when} & \mathtt{left} & \textbf{do} & \ulcorner \mathrm{exch}_{\mathtt{prd}}[x, x'] \urcorner). \end{array}$$

The process $Pred_i$ with $i \in \{0, 1\}$ models the behavior of predator i. The operator $\oplus$ denotes binary summation.

$$
\begin{array}{llllll}
Pred_i & \stackrel{\text{def}}{=} ! & (& \textbf{when } x_i = x & \textbf{do}\,(\ulcorner l := i \urcorner \parallel \ulcorner Pursuit_i \urcorner) & \\
& & + & \textbf{when } l = i \wedge x_{i\oplus 1} \neq x & \textbf{do}\,\ulcorner Pursuit_i \urcorner & \\
& & + & \textbf{when } l = i \oplus 1 \wedge x_i \neq x & \textbf{do}\,\ulcorner Support_i \urcorner &).
\end{array}
$$

Thus whenever $Pred_i$ is in front of the prey (i.e. $x_i = x$) it declares itself as the leader by assigning i to the cell l. Then it runs process $Pursuit_i$ defined below and keeps doing it until the other predator $Pred_{i\oplus 1}$ declares itself the leader. If the other process is the leader then $Pred_i$ runs process $Support_i$ defined below.

Process $Pursuit_i$, whenever the prey is above of corresponding predator $(y_i < y \wedge x_i = x)$, tells the other predator that the move is to go up and increases by two the contents of y_i while keeping in cell y'_i the previous value. The other cases which correspond to going left, right and down can be described similarly.

$$
\begin{array}{lllll}
Pursuit_i & \stackrel{\text{def}}{=} & \textbf{when} & (y_i < y \wedge x_i = x) & \textbf{do} \\
& & & (\ulcorner \text{exch}_{\texttt{succ}^2}[y_i, y'_i]\urcorner \quad \parallel & \textbf{tell}(\texttt{up}_i)) \\
& & +\,\textbf{when} & (y_i > y \wedge x_i = x) & \textbf{do} \\
& & & (\ulcorner \text{exch}_{\texttt{prd}^2}[y_i, y'_i]\urcorner \quad \parallel & \textbf{tell}(\texttt{down}_i)) \\
& & +\,\textbf{when} & (x_i < x \wedge y_i = y) & \textbf{do} \\
& & & (\ulcorner \text{exch}_{\texttt{succ}^2}[x_i, x'_i]\urcorner \quad \parallel & \textbf{tell}(\texttt{right}_i)) \\
& & +\,\textbf{when} & (x_i > x \wedge y_i = y) & \textbf{do} \\
& & & (\ulcorner \text{exch}_{\texttt{prd}^2}[x_i, x'_i]\urcorner \quad \parallel & \textbf{tell}(\texttt{left}_i)).
\end{array}
$$

The process $Support_i$ is defined according to the move decision of the leader. Hence, if the leader moves up (e.g. $\texttt{up}_{i\oplus 1}$) then the support predator moves up as well. The other cases are similar.

$$
\begin{array}{lllll}
Support_i & \stackrel{\text{def}}{=} & \textbf{when} & \texttt{up}_{i\oplus 1} & \textbf{do} \\
& & & (\ulcorner \text{exch}_{\texttt{succ}^2}[y_i, y'_i]\urcorner \quad \parallel & \textbf{tell}(\texttt{up}_i)) \\
& & +\,\textbf{when} & \texttt{down}_{i\oplus 1} & \textbf{do} \\
& & & (\ulcorner \text{exch}_{\texttt{prd}^2}[y_i, y'_i]\urcorner \quad \parallel & \textbf{tell}(\texttt{down}_i)) \\
& & +\,\textbf{when} & \texttt{right}_{i\oplus 1} & \textbf{do} \\
& & & (\ulcorner \text{exch}_{\texttt{succ}^2}[x_i, x'_i]\urcorner \quad \parallel & \textbf{tell}(\texttt{right}_i)) \\
& & +\,\textbf{when} & \texttt{left}_{i\oplus 1} & \textbf{do} \\
& & & (\ulcorner \text{exch}_{\texttt{prd}^2}[x_i, x'_i]\urcorner \quad \parallel & \textbf{tell}(\texttt{left}_i)).
\end{array}
$$

We assume that initially $Pred_0$ is the leader and that it is in the first row in the middle column . The other predator is next to it in the same row. The prey is just above $Pred_0$. The process $Init$ below specifies these conditions. Let $p = max/2$.

$$
\begin{array}{lll}
Init & \stackrel{\text{def}}{=} & \prod_{i\in 0,1}(\ulcorner x_i : (p+i)\urcorner \parallel \ulcorner y_i : (0)\urcorner \parallel \ulcorner x'_i : (p+i)\urcorner \parallel \ulcorner y'_i : (0))\urcorner \\
& & \parallel \ulcorner x : (p)\urcorner \parallel \ulcorner y : (1)\urcorner \parallel \ulcorner x' : (p)\urcorner \parallel \ulcorner y'_i : (1)\urcorner \parallel \ulcorner l : 0\urcorner.
\end{array}
$$

Operationally, one can verify that

$$\ulcorner \mathit{Init} \urcorner \parallel \ulcorner \mathit{Pred}_0 \urcorner \parallel \ulcorner \mathit{Pred}_1 \urcorner \parallel \ulcorner \mathit{Prey} \urcorner \models \Diamond(\mathtt{capture}_0 \,\dot{\vee}\, \mathtt{capture}_1)$$

thus stating that the predators eventually capture the prey under our initial conditions. Notice we only used locally independent processes. Thus the proposition below follows from the completeness of the inference system.

Proposition 7.4. (A Temporal Property of Predators)

$$\ulcorner \mathit{Init} \urcorner \parallel \ulcorner \mathit{Pred}_0 \urcorner \parallel \ulcorner \mathit{Pred}_1 \urcorner \parallel \ulcorner \mathit{Prey} \urcorner \vdash \Diamond(\mathtt{capture}_0 \,\dot{\vee}\, \mathtt{capture}_1).$$

It is worth noticing that in the case of one single predator, say Pred_0, the prey may sometimes escape under the same initial conditions, i.e. $\ulcorner \mathit{Init} \urcorner \parallel \ulcorner \mathit{Pred}_0 \urcorner \parallel \ulcorner \mathit{Prey} \urcorner \nvdash \Diamond \mathtt{capture}_0$. A similar situation occurs if the predators were not allowed to alternate the leader role.

7.5 Musical Applications

In the last decade several formalisms have been proposed to account for musical structures and the operations used to construct and transform them ([Barbar *et al.* 1998], [Balaban and Samoun 1993], [Pachet *et al.* 1996]. We can regard music performance and composition as a complex task of defining and controlling interaction among concurrent activities. In [Alvarez *et al.* 2001], *PiCO*, a concurrent processes calculus integrating constraints and objects was proposed. Musical applications are programmed in a visual language having this calculus as its underlying model. Since there is no explicit notion of time in *PiCO* some musical examples, in particular those involving time and synchronization, are difficult to express. In this section we model two of those examples.

In the following example we shall use the derived construct

$$W_{(c,P)} \stackrel{\text{def}}{=} \textbf{when } c \textbf{ do } P \;\parallel\; \textbf{unless } c \textbf{ next } W_{(c,P)}.$$

This construct waits until c holds and then executes P. We use the more readable notation **whenever** c **do** P for $W_{(c,P)}$.

7.5.1 Music Example: Controlled Improvisation.

This example models a controlled improvisation musical system. There is a certain number m of *musicians* (or *voices*), each playing blocks of three notes. Each of them is given a particular pattern (i.e., a list) of allowed delays between each note in the block. The musician can freely choose any permutation of his pattern. For example, given a pattern $p = [4, 3, 5]$ a musician can play his block with spaces of 5 then 4 and then 3 between the notes. Once a musician has finished playing his block of three notes, he must wait for a signal of the *conductor* telling him that the others musicians have also finished their respective blocks. Only after this he can start playing a new block. The exact time in which he actually starts playing a new block is not specified, but it is constrained to be no later than the sum of the durations of all patterns. For example, for three musicians and patterns $p_1 = [3,2,2]$, $p_2 = [4,3,5]$ and $p_3 = [3,3,4]$ no delay between blocks greater than 29 time units is allowed. Musicians keep playing this way until all of them play a note at the same time. After this, all the musicians should stop playing.

In order to model this example we assume that constant $\mathtt{sil} \in \mathcal{D}$ represents some note value for silence. Process M_i, $i \leq m$, models the activity of the i−th musician. When ready to start playing ($start_i = 1$), the i−th musician chooses a permutation (j,k,l) of his given pattern p_i. Then, M_i spawns a process $Play^i_{(j,k,l)}$, thus playing a note at time j (after starting), but not before, then at time $j+k$ but not before, and finally at time $j+k+l$. Constraint $c_i[note_i]$ specifies some value for $note_i$ different from $\mathtt{sil}$. After playing his block, the i−th musician signals termination by setting cell $flag_i$ to 1. Furthermore, upon receiving the $go = 1$ signal, the i−th musician eventually starts a new block no later than $\mathtt{pdur}$ which is a constant representing the sum of the durations of all patterns.

$$
\begin{aligned}
M_i \stackrel{\text{def}}{=}\ & \mathbf{!when}\ (start_i = 1)\ \mathbf{do} \\
& \sum_{(j,k,l)\in perm(p_i)} \Big(\ulcorner Play^i_{(j,k,l)} \urcorner \parallel \ \mathbf{next}^{j+k+l}\big(\ulcorner flag_i := 1 \urcorner \parallel \\
& \qquad\qquad\qquad\qquad\qquad \mathbf{whenever}\ (go = 1)\ \mathbf{do} \\
& \qquad\qquad\qquad\qquad\qquad\quad \star_{[0,\mathtt{pdur}]}\mathbf{tell}(start_i = 1)\big)\Big)
\end{aligned}
$$

$$
\begin{aligned}
& Play^i_{(j,k,l)} \stackrel{\text{def}}{=} \\
& \ !_{[0,j-1]}\mathbf{tell}(note_i = \mathtt{sil}) \ \parallel\ \mathbf{next}^{j}\mathbf{tell}(c_i[note_i]) \\
& \parallel\ !_{[j+1,j+k-1]}\mathbf{tell}(note_i = \mathtt{sil}) \ \parallel\ \mathbf{next}^{j+k}\mathbf{tell}(c_i[note_i]) \\
& \parallel\ !_{[j+k+1,j+k+l-1]}\mathbf{tell}(note_i = \mathtt{sil}) \ \parallel\ \mathbf{next}^{j+k+l}\mathbf{tell}(c_i[note_i])
\end{aligned}
$$

The *Conductor* process is always checking (listening) whether all the musicians play a note exactly at the same time $\bigwedge_{i\in[1,m]}(note_i \neq \mathtt{sil})$. If this happens it sets the cell *stop*, initially set to 0, to 1. At the same time, it waits for all flags to be set to 1, and then resets the flags and gives the signal $go = 1$ to all musician to start a new block, unless all of them have output a note at the same time (i.e., $stop = 1$).

$$
\begin{aligned}
& Conductor \stackrel{\text{def}}{=} \\
& \ \mathbf{whenever} \bigwedge_{i\in[1,m]} (note_i \neq \mathtt{sil})\ \mathbf{do}\ \ulcorner stop := 1 \urcorner \\
& \parallel \mathbf{!when} \bigwedge_{i\in[1,m]} (flag_i = 1) \wedge (stop = 0)\ \mathbf{do}\ (\mathbf{tell}(go = 1) \parallel \prod_{i\in[1,m]} \ulcorner flag_i := 0 \urcorner)
\end{aligned}
$$

Initially the m flag cells are set to 0, the M_i are given the start signal $start_i = 1$ and, as mentioned above, the cell *stop* is set to 0. The system (i.e., the performance) *System* is just the parallel execution (performance) of all the M_i musicians controlled by the *Conductor* process.

$$Init \stackrel{\text{def}}{=} \prod_{i\in[1,m]}(\ulcorner flag_i : 0\urcorner \parallel \mathbf{tell}(start_i = 1)) \parallel \ulcorner stop : 0\urcorner$$

$$System \stackrel{\text{def}}{=} \ulcorner Init\urcorner \parallel \ulcorner Conductor\urcorner \parallel \prod_{i\in[1,m]}\ulcorner M_i\urcorner$$

The temporal logic of ntcc can then be used to formally specify and prove termination properties for this system. For example, we may wonder whether the assertion

$$\ulcorner System\urcorner \vdash \Diamond stop = 1$$

holds. This assertion expresses that the musicians eventually stop playing at all regardless their choices. We may also wonder whether there exists certain choices of musicians for which they eventually stops playing note at all. For proving this we can verify whether the assertion

$$\ulcorner System\urcorner \vdash \Box stop = 0$$

does not hold, i.e., there is a run of the system for which at some time unit all the notes are different from `sil`. Since $\ulcorner System\urcorner$ is locally independent, it follows from completeness result that these termination properties can be formally proved in the inference system for ntcc.

7.5.2 Rhythm Patterns

In this section we shall model synchronization of rhythm patterns in ntcc. Let us first define a "metronome" process.

$$\begin{array}{l} M[tick, count, \delta] \stackrel{\text{def}}{=} \\ \quad !(\ \mathbf{when}\ (count \ \mathtt{mod}\ \delta) = 0 \ \mathbf{do}\ \ulcorner tick := tick + 1\urcorner \parallel \ulcorner count := 0\urcorner \\ \quad\ \ + \mathbf{when}\ (count \ \mathtt{mod}\ \delta) \neq 1 \ \mathbf{do}\ \ulcorner count := count + 1\urcorner) \end{array}$$

One could think of $M[tick, count, \delta]$ as a process that "ticks" (by increasing *tick*) every δ time units. This process could be controlled by the acceleration process:

$$Accel[signal, \delta] \stackrel{\text{def}}{=} !\ \mathbf{when}\ signal = 1 \wedge \delta > 0 \ \mathbf{do}\ \ulcorner \delta := \delta - 1\urcorner$$

Process $Accel[signal, k]$ can "speed up the ticks of $M[tick, count, \delta]$" by decreasing δ, if some other process, which we shall refer to as $Control[signal]$, tells $signal = 1$.

We can now define the Rythm process $R_{(s,d,e)}[tick, note]$ which can be synchronized by $M[tick, count, \delta]$ and thus possibly accelerated by $Control[signal]$.

$$R_{(s,d,e)}[tick, note] \stackrel{\text{def}}{=}$$
$$!\ \mathbf{when}\ s >= tick >= e \wedge (tick - start)\ \mathtt{mod}\ d = 0\ \mathbf{do}\ Pitch[note]$$

Process $R_{(s,d,e)}[tick, note]$ runs a certain $Pitch[note]$ process, which outputs some pitch on $note$, at every $d$uration-th tick, from the $s$tart-th tick to the end-th tick. Adding rhythms of two eight notes, two triplets and two quintuplets can then be defined by the system:

$$System \stackrel{\text{def}}{=}$$
$$\mathbf{local}\ tick, \delta, count, signal\ \mathbf{in}\ ($$
$$Init \parallel Control[signal] \parallel \ulcorner M[tick, count, \delta]\urcorner \parallel \ulcorner Accel(signal, \delta)\urcorner$$
$$\parallel \ulcorner R_{(0,30,120)}[tick, \delta]\urcorner \parallel \ulcorner R_{(0,20,120)}[tick, \delta]\urcorner \parallel \ulcorner R_{(0,12,120)}[tick, \delta]\urcorner\ )$$

where $Init = \ulcorner tick : 0\urcorner \parallel \ulcorner \delta : 30\urcorner \parallel \ulcorner count : 0\urcorner$.

Since the *Rhythm* processes depend on variables $tick$ and δ, complex patterns of interactions of global and local speeds, such as metric modulations, can be modeled.

8. Related Work

Our proposal is a strict extension of tcc [Saraswat *et al.* 1994], 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 nondeterminism or unbounded finite-delay. In [Saraswat *et al.* 1994] 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 [Manna and Pnueli 1991] and is complete for locally independent ntcc processes, hence also for tcc processes.

An extension of tcc, which does not consider nondeterminism or unbounded finite-delay, has been proposed in [Saraswat *et al.* 1996]. 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 [de Boer *et al.* 2000], in the *specification* of (large) timed systems weak pre-emption often suffices (and nondeterminism 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 [Reiter 1980].

Other extensions of tcc have been proposed in Gupta *et al.* [1998, 1999]. In [Gupta *et al.* 1998] processes can evolve continuously as well as discretely. The language in [Gupta *et al.* 1999] allows random assignments with some given distribution. None of these extensions, however, consider nondeterminism or unbounded finite-delay.

The tccp calculus [de Boer *et al.* 2000] is the only other proposal for a nondeterministic 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 [de Boer *et al.* 2002]. The underlying linear temporal logic in [de Boer *et al.* 2002] 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 [Manna and Pnueli 1991]. This may come in handy when using the Consequence Rule which is also present in [de Boer *et al.* 2002].

Functional Reactive Programming [Wan and Hudak 2000] (FRP) and Temporal Logic Programming [Moszkowski 1986] (TLP) are high-level declarative frameworks for programming reactive systems. In FRP programs are described in terms of continuous, time-varying, reactive values, and conditions that occur at discrete points in time. In TLP programs are described in terms of temporal formulae. The main distinction between ntcc and these frameworks arises from the underlying paradigm upon they are defined. In ntcc the underlying paradigm is concurrent constraint programming whereas in FRP is functional programming and in TLP is logic programming. As argued in [Saraswat *et al.* 1994], in comparison with logic programming ccp provides a more algebraic view of process combinators to model concurrency. Such an algebraic view is fundamental in concurrency theory [Milner 1980]. FRP does not deal with algebraic issues for concurrency. In fact, only recently FRP was extended with nondeterminism to provide for parallel programming [Peterson *et al.* 2000].

The ntcc calculus has also been studied in [Nielsen and Valencia 2002] and [Nielsen *et al.* 2002]. In [Nielsen and Valencia 2002] we studied in detail the equivalences arising from the observable behavior introduced in Section 4. In particular it is shown that these equivalences and their induced congruences are decidable for a substantial fragment of the calculus. In [Nielsen *et al.* 2002] we compared the expressive power of several variants of the calculus. These variants differ mainly in their way of expressing infinite behavior. In particular it is shown that general recursion is strictly more expressive than replication which as expressive as parameterless recursion with static scoping. Parameterless recursion with dynamic scoping, however, is as expressive as general recursion.

9. Concluding Remarks.

We introduced a model of temporal concurrent constraint programming which we call the ntcc calculus. We showed examples of its applicability to timed systems, multi-agent systems and musical applications. We provided a denotational semantics that approximates the notion of strongest postcondition of processes (in the sense of [de Boer *et al.* 1997]) and identified an significant fragment for which such a denotation is complete. We defined a linear temporal logic following the standard definition of Manna and Pnueli [1991], 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 inference system for proving that a process satisfies a linear-temporal specification, and we applied it to prove some temporal properties of our examples.

Our current research in ntcc includes the study of decidability issues of the input-output equivalence for ntcc. From this study, we learned that the combination between nondeterminism and hiding (even without the finite-delay operator) presents technical difficulties as one can express unbounded nondeterministic processes with them.

In the search of a fully-abstract fixed point model with respect to the input-output behavior, we found that although ntcc allows countable nondeterminism and infinite computations to happen (see [Apt and Plotkin 1986] and [Nyström 1996] 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 [Herescu and Palamidessi 2000]. This is justified by the existence of RCX program examples involving stochastic behavior which cannot be faithfully modeled with nondeterministic behavior. We also also plan to study decidability issues for our logic. Namely, whether a formulae is valid and whether a given process satisfies a given temporal formula. In a more practical setting we plan to define a programming language for RCX controllers based on ntcc.

Acknowledgements

The authors thank Maurizio Gabbrielli, Camilo Rueda, Dale Miller, Vineet Gupta, Radha Jagadeesan, Prakash Panangaden, and Marco Carbone for helpful comments and criticism on various aspects of ntcc. Thanks also goes to the anonymous referees for their valuable suggestions on how to improve this paper.

References

Alvarez, G., Diaz, J.F., Quesada, L.O., Rueda, C., Tamura, G., Valencia, F.D., and Assayag, G. 2001. Integrating Constraints and Concurrent Objects in Musical Applications: A Calculus and its Visual Language. *Constraints 6*, 1, 21–25.

Apt, K. and Plotkin, G. 1986. Countable Nondeterminism and Random Assignment. *JACM 33*, 724–767.

Balaban, M. and Samoun, C. 1993. Hierarchy, Time and Inheritance in Music Modelling. *Languages of Design 1*, 2, 147–172.

Barbar, K., Beurivé, A., and Desainte-Catherine, M. 1998. Structures hierarchiques pour la composition assisté par ordinateur. In *Recherches et applications en informatique musicale*, Chemillier, M. and Pachet, F., Editors. Collection Informatique Musicale. HERMES Science, chapter 8, 109–124.

Benda, M., Jagannathan, V., and Dodhiawala, R. 1986. On Optimal Cooperation of Knowledge Sources: An Empirical Investigation. Tech. Report BCS-G2010-28, Boeing Advanced Technology Center.

Berry, G. and Gonthier, G. 1992. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science of Computer Programming 19*, 2, 87–152.

de Boer, F., Gabbrielli, M., and Meo, M. C. 2000. A Timed Concurrent Constraint Language. *Information and Computation 161*, 1, 45–83.

de Boer, F., Gabbrielli, M., and Meo, M. C. 2002. Proving Correctness of Timed Concurrent Constraint Programs. In *Proc. of 5th International Conference on Foundations of Software Science and Computation Structures*, Volume 2303 of *LNCS*. Springer-Verlag, 37–51.

de Boer, F. S., Gabbrielli, M., Marchiori, E., and Palamidessi, C. 1997. Proving Concurrent constraint Programs Correct. *ACM Transactions on Programming Languages and Systems 19*, 5, 685–725.

Falaschi, M., Gabbrielli, M., Marriott, K., and Palamidessi, C. 1997. Confluence in Concurrent Constraint Programming. *Theoretical Computer Science 183*, 2, 281–315.

Fredslund, J. 2000. The Assumption Architecture. Progress Report, University of Aarhus. Available via http://www.daimi.au.dk/∼chili/publications.html.

Gupta, V., Jagadeesan, R., and Panangaden, P. 1999. Stochastic Processes as Concurrent Constraint Programs. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM Press, 189–202.

Gupta, V., Jagadeesan, R., and Saraswat, V. A. 1998. Computing with continuous change. *Science of Computer Programming 30*, 1–2, 3–49.

Halbwachs, N. 1998. Synchronous Programming of Reactive Systems. In *Proc. of the 10th International Conference on Computer Aided Verification*, Volume 1427 of *LNCS*. Springer-Verlag, 1–16.

Haynes, T. and Sen, S. 1996. Evolving Behavioral Strategies in Predators and Prey. In *Proc. of the IJCAI Workshop on Adaption and Learning in Multi-Agent Systems*, Volume 1042 of *LNAI*. Springer-Verlag, 113–126.

Herescu, M. and Palamidessi, C. 2000. Probabilistic Asynchronous π-Calculus. In *Proc. of the Second International Conference on Foundations of Software Science and Computation Structures*, Volume 1784 of *LNCS*. Springer-Verlag, 146–160.

Hodkinson, I., Wolter, F., and Zakharyaschev, M. 2000. Decidable Fragments of First-Order Temporal Logics. *Annals of Pure and Applied Logic 106*, 1–3, 85–134.

Jaffar, J. and Lassez, J. L. 1987. Constraint Logic Programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 111–119.

Lund, H. H. and Pagliarini, L. 1999. Robot Soccer with LEGO Mindstorms. In *RoboCup*, Volume 1604 of *LNCS*. Springer-Verlag, 141–151.

Manna, Z. and Pnueli, A. 1991. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag.

Milner, R. 1980. *A Calculus of Communicating Systems*. Springer-Verlag.

Milner, R. 1992. A finite delay operator in Synchronous CCS. Tech. Report CSR-116-82, University of Edinburgh.

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

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

Niehren, J. and Smolka, G. 1994. A Confluent Relational Calculus for Higher-Order Programming with Constraints. In *First International Conference on Constraints in Computational Logics*, Volume 845 of *LNCS*. Springer-Verlag, 89–104.

Nielsen, M., Palamidessi, C., and Valencia, F. 2002. On the Expressive Power of Concurrent Constraint Programming Languages. Tech. Report RS-02-22, BRICS, University of Aarhus.

Nielsen, M. and Valencia, F. 2002. Temporal Concurrent Constraint Programming: Applications and Behavior. In *Formal and Natural Computing: Essays Dedicated to Grzegorz Rozenberg*. Volume 2300 of *LNCS*. Springer-Verlag, chapter 4, 298–324.

Nolfi, S. and Floreano, D. 1998. Coevolving Predator and Prey Robots: Do "Arms Races" Arise in Artificial Evolution? *Artificial Life 4*, 4, 311–335.

Nyström, S.O. 1996. There is no Fully Abstract Fixpoint Semantics for Non-Deterministic Languages with Infinite Computations. *Information Processing Letters 60*, 6, 289–293.

Pachet, F., Ramalho, G., and Carrive, J. 1996. Representing Temporal Musical Objects and Reasoning in the MusES System. *Journal of New Music Research 25*, 3, 252–275.

Palamidessi, C. and Valencia, F. 2001. A Temporal Concurrent Constraint Programming Calculus. In *Proc. of the 7th International Conference on Principles and Practice of Constraint Programming*, Volume 2239 of *LNCS*. Springer-Verlag, 302–316.

Peterson, J., Trifonov, V., and Serjantov, A. 2000. Parallel Functional Reactive Programming. In *Practical Aspects of Declarative Languages*, Volume 1753 of *LNCS*. Springer-Verlag, 16–31.

Reiter, R. 1980. A Logic for Default Reasoning. *Artificial Intelligence 13*, 1–2, 81–132.

Saraswat, V. 1993. *Concurrent Constraint Programming*. The MIT Press.

Saraswat, V., Jagadeesan, R., and Gupta, V. 1994. Foundations of Timed Concurrent Constraint Programming. In *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society Press, 71–80.

Saraswat, V., Jagadeesan, R., and Gupta, V. 1996. Timed Default Concurrent Constraint Programming. *Journal of Symbolic Computation 22*, 5–6, 475–520.

Saraswat, V., Rinard, M., and Panangaden, P. 1991. The Semantic Foundations of Concurrent Constraint Programming. In *Proc. of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*. ACM Press, 333–352.

Saraswat, Vijay, Jagadeesan, Radha, and Gupta, Vineet. 1994. Programming in Timed Concurrent Constraint Languages. In *Constraint Programming*, Mayoh, B., Tyugu, E., and Penjaam, J., Editors. Volume 131 of *NATO ASI Sub-series F: Computer and System Sciences*. Springer-Verlag, chapter 4, 361–410.

Shapiro, E. 1990. The Family of Concurrent Logic Programming Languages. *Computing Surveys 21*, 3, 413–510.

Stone, P. and Veloso, M. 2000. Multiagent Systems: A Survey from a Machine Learning Perspective. *Autonomous Robots 8*, 345–383.

Wan, Z. and Hudak, P. 2000. Functional Reactive Programming from First Principles. In *Proceedings of the ACM SIGPLAN '00 conference on Programming language design and implementation*, Volume 35 of *SIGPLAN Notices*. ACM Press.

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