BRICS

Basic Research in Computer Science

BRICS RS-01-48 Nielsen & Valencia: Temporal Concurrent Constraint Programming: Applications and Behavior

Temporal Concurrent Constraint Programming

Applications and Behavior

Mogens Nielsen
Frank D. Valencia

BRICS Report Series RS-01-48
ISSN 0909-0878 December 2001

Copyright © 2001, Mogens Nielsen & Frank D. Valencia.
BRICS, Department of Computer Science
University of Aarhus. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.

See back inner page for a list of recent BRICS Report Series publications.
Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK–8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through the World Wide Web and anonymous FTP through these URLs:

http://www.brics.dk
ftp://ftp.brics.dk
This document in subdirectory RS/01/48/

Temporal Concurrent Constraint Programming: Applications and Behavior

Mogens Nielsen Frank D. Valencia

December, 2001

Abstract

The *ntcc* calculus is a model of non-deterministic temporal concurrent constraint programming. In this paper we study behavioral notions for this calculus. In the underlying computational model, concurrent constraint processes are executed in discrete time intervals. The behavioral notions studied reflect the reactive interactions between concurrent constraint processes and their environment, as well as internal interactions between individual processes. Relationships between the suggested notions are studied, and they are all proved to be decidable for a substantial fragment of the calculus. Furthermore, the expressive power of this fragment is illustrated by examples.

1 Introduction

Concurrent constraint programming [19] has been studied extensively as a paradigm for specifying and programming reactive systems. One of the main features of ccp is that it is based on a declarative as well as operational computational model.

The fundamental primitive of a *constraint* is a partial information on values of variables (e.g. $x + y > 5$). The state of a computation (also called a *store*) is simply a set of constraints, and during a computation, a process may modify the state by telling information. Also, a process may condition its activity by asking for certain information to be entailed by the present store - operationally blocking its activity until other processes provide the requested information (if ever). In this way *concurrent* processes may communicate via the common store of constraints. Processes in ccp are built using the basic primitives of telling and asking constraints, and the operators of parallel composition, hiding and recursion.

The *temporal* ccp computational model introduced in [20] is an extension aimed at specifying timed systems following the paradigms of Synchronous Languages ([2]). Time is conceptually divided into discrete intervals (or time units). In a particular time interval, a ccp process receives a stimulus (i.e. a constraint) from the environment, it executes with this stimulus as the initial store, and when it reaches its resting point, it responds to the environment with the resulting store. Also the resting point determines a residual process, which is then executed in the next time interval.

This temporal ccp model is inherently deterministic. In [17] a non-deterministic version of the calculus was introduced, adding e.g. (non-deterministic) guarded choice and unbounded-finite delay as new operators in the language of processes. The extension was argued to be consistent with the declarative flavor of ccp, i.e. to free the programmer from over-specifying a deterministic solution, when a non-deterministic simple solution is more appropriate (following the arguments behind Dijkstra's language of guarded commands). Furthermore, it was argued that a very important benefit of allowing the specification of non-deterministic behavior arises when modeling the interaction among several components running in parallel, in which one component is part of the environment of the others. These systems often need non-determinism to be modeled faithfully.

In this paper we introduce and study various notions of behavior for the *ntcc* calculus: the input-output and the language equivalence and their congruences, all motivated operationally and/or logically. The notions are related, and they are all proved to be decidable for a substantial fragment of the calculus. The decidability for the complete calculus is left open.

Furthermore, we illustrate the expressive power of our fragment of *ntcc* by modeling constructs such as cells and some applications involving the programming of RCX$^{\mathrm{TM}}$ controllers, and a version of a Predator/Prey (Pursuit) game.

2 The Calculus

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

2.1 Constraint Systems

Concurrent constraint languages are parameterized by a *constraint system.* Basically, a constraint system defines the underlying universe of the particular language. It provides a signature from which syntactically denotable objects in language called *constraints* can be constructed, and an entailment relation specifying interdependencies between such constraints. For our purposes it will suffice to consider the notion of constraint system based on First-Order Predicate Logic, as it was done in [24][1]

Definition 2.1 *A* constraint system *is a pair* (Σ, Δ) *where* Σ *is a signature specifying functions and predicate symbols, and* Δ *is a consistent first-order theory.*

Given a constraint system (Σ, Δ), let $\mathcal{L}$ be the underlying first-order language $(\Sigma, \mathcal{V}, \mathcal{S})$, where $\mathcal{V}$ is a countable set of variables and $\mathcal{S}$ is the set of logical symbols $\wedge$, $\vee$, $\Rightarrow$, $\neg$, $\exists$, `true` and `false` which denote logical conjunction, disjunction, implication, negation, existential quantification and the always true and 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 $c \vdash d$, if the formula $c \Rightarrow d$ holds in all models of Δ. We shall require $\vdash$ to be decidable. We say that c is equivalent to d, written $c \approx d$, iff $c \vdash d$ and $d \vdash c$. We define the (relevant) *free-variables* of c as $fv(c) = \{x \in \mathcal{V} \mid \exists_x c \not\approx c\}$ (e.g., $fv(x = x \wedge y > 1) = \{y\}$).

Henceforth, $\mathcal{C}$ is a set of constraints modulo $\approx$ in (Σ, Δ). The set $\mathcal{C}$ is closed wrt conjunction and existential quantification and it represents the constraints under consideration in the underlying constraint system.

[1]See [22] for a more general notion of constraints based on Scott's information systems.

2.2 Process 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 by the following syntax:

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

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

Process $P \parallel Q$ represents the parallel composition of P and Q. In one time unit (or interval) P and Q operate concurrently, "communicating" via the common store. We use $\prod_{i \in I} P_i$, where I is finite, to denote the parallel composition of all P_i. Process **local** x **in** P behaves like P, except that all the information on x produced by P can only be seen by P.

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

The operator "!" is a delayed version of the replication operator for the $\pi-$calculus ([15]): $!\, P$ 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.

Our process language is essentially the language of the calculus ntcc from [17], but in order to unify and to simplify the presentation of our technical results, we have omitted the unbounded finite delay operator. As we shall clarify, it is not clear to what extent all our results generalize to the full language of ntcc.

2.3 An Operational Semantics.

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

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

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

1. $(Proc/_{\equiv}, \|, \mathbf{skip})$ *is a symmetric monoid.*
2. $P \equiv Q$ *if they only differ by a renaming of bound variables.*
3. $\mathbf{next}\,\mathbf{skip} \equiv \mathbf{skip} \qquad \mathbf{next}(P \parallel Q) \equiv \mathbf{next}\,P \parallel \mathbf{next}\,Q.$
4. $\mathbf{local}\,x\,\mathbf{in}\,\mathbf{skip} \equiv \mathbf{skip} \quad \mathbf{local}\,x\,y\,\mathbf{in}\,P \equiv \mathbf{local}\,y\,x\,\mathbf{in}\,P.$
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 reduction relations $\longrightarrow\ \subseteq \Gamma \times \Gamma$ and $\Longrightarrow\ \subseteq Proc \times \mathcal{C} \times \mathcal{C} \times Proc$ are the least relations satisfying the rules appearing in Table 1. The *internal transition* $\langle P, c\rangle \longrightarrow \langle Q, d\rangle$ should be read as "P with store c reduces, in one internal step, to Q with store d ". The *observable transition* $P \xRightarrow{(c,d)} Q$ should be read as "P on input c reduces, in one time unit, to Q with store d ". As in tcc, the store does not transfer automatically from one interval to another.

We now give a description of the operational rules. Rules TELL, CHOICE, PAR and LOC are standard [22]. Rule UNLESS says that if c is entailed by the current store, then the execution of the process P (in the next time interval) is precluded. Rule REPL specifies that the process $!\,P$ produces a copy P at the current time unit, and then persists in the next time unit. Rule STRUCT simply says that structurally congruent processes have the same reductions.

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

Definition 2.3 (Future Function) *The partial function $F : Proc \rightharpoonup Proc$ is defined as follows:*

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

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

2.3.1 Interpreting Processes Runs.

Henceforward we use α, α' to represent elements of $\mathcal{C}^\omega$. 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 environment. At the time unit i, the environment provides a *stimulus* c_i and P_i produces c_i' as *response*. 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. We shall refer to the set of all timed observations of a process P as the *language* of P.

In section 4 we study in detail input-output behavior and language of processes.

Notation 1 *Throughout the paper we use the following notation on transitions:*

1) $P \longrightarrow Q$ *iff* *for some* c, $\langle P, c\rangle \longrightarrow \langle Q, c'\rangle$.
2) $P \Longrightarrow Q$ *iff* $P \longrightarrow^* P' \not\longrightarrow$ *and* $Q = F(P')$.
3) $P \overset{c}{\Longrightarrow} Q$ *iff* $P \xRightarrow{(\mathtt{true},c)} Q$.
4) $P \overset{\alpha}{\Longrightarrow}{}^\omega$ *iff* $P \xRightarrow{(\mathtt{true}^\omega,\alpha)}{}^\omega$.

2.4 A Logic of ntcc Processes

A relatively complete formal system for proving whether or not an ntcc process satisfies a linear-temporal property was introduced in [17]. In this section we summarize these results.

We extend the ccp notion of strongest postcondition of a process P ([6]), $sp(P)$, to our setting. In ntcc, $sp(P)$ denotes the set of all infinite sequences that P can possibly output. More precisely,

Definition 2.4 *Given* P *its* strongest postcondition *is defined as*

$$sp(P) = \{\alpha' \mid \text{ for some } \alpha : P \xRightarrow{(\alpha,\alpha')}{}^\omega\}.$$

2.4.1 Temporal Logic.

We define a linear temporal logic for expressing properties of ntcc processes. The formulae $A, B, ... \in \mathcal{A}$ are defined by the grammar

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

where c denotes an arbitrary constraint. 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 semantics of the logic is given in Definition 2.5. The standard interpretation structures of linear temporal logic are infinite sequences of states [14]. In the case of ntcc, states are represented by constraints, thus we consider as interpretations the elements of $\mathcal{C}^\omega$.

Definition 2.5 *We say that $\alpha \in \mathcal{C}^\omega$ is a model of A, notation $\alpha \models A$, if $\langle \alpha, 1 \rangle \models A$, where:*

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

where $\exists_x \alpha$ represents the sequence obtained by applying $\exists_x$ to each constraint in α. 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 shall say that P *satisfies* A iff every infinite sequence that P can possibly output satisfies the property expressed by A, i.e. $sp(P) \subseteq [\![A]\!]$. A relatively complete proof system for assertions $P \;\vdash\; A$, whose intended meaning is that P satisfies A, can be found in [17]. We shall write $P \;\vdash\; A$ if there is a derivation of $P \;\vdash\; A$ in this system.

3 Applications

Let us assume that the underlying constraint system is $FD[max]$ which has $\{\texttt{succ}, \texttt{prd}, +, \times, =, <, >, 0, 1, \ldots\}$ as signature and the set of sentences valid in arithmetic modulo max as theory. Henceforth, we designate Dom as the set $\{0, 1, ..., max-1\}$ and use v and w to range over its elements.

It will be convenient to specify our applications using defining equations of the form $q(x_1, \ldots, x_m) \stackrel{\text{def}}{=} P_q$. In ntcc we encode definitions of this sort provided that P_q contains at most one occurrence of q which must be within the scope of a "**next**" and out of the scope of any "!". The reason for such a restriction is that we want to keep the response time of the system bounded: we do not want P_q to make unboundely many recursive calls within a time interval. The intended behavior of a call of q with arguments $t_1, \ldots, t_m$, written $\ulcorner q(t_1, \ldots, t_m)\urcorner$, when $t_i = v_i$ in the current store, is that of $P_q[v_1/x_1, \ldots, v_m/x_m]$ [2]. The encoding of a process definition requires the use of replication and, if the definition is recursive or it has at least one parameter, also hiding (see [18] for the exact details of the encoding).

3.1 Cell Example

Cells provide a basis for the specification and analysis of mutable and persistent data structures as shown for the π calculus. We 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 can, in the future, be assigned a new value.

$$
\begin{array}{lcll}
x\colon(z) & \stackrel{\text{def}}{=} & \mathbf{tell}(x = z) \parallel \mathbf{unless}\ \mathtt{change}(x)\ \mathbf{next}\ x\colon(z) & \\
g_{\text{exch}}(x,y) & \stackrel{\text{def}}{=} & \sum_{v} \mathbf{when}\ (x = v)\ \mathbf{do}\ (& \mathbf{tell}(\mathtt{change}(x)) \parallel \mathbf{tell}(\mathtt{change}(y)) \\
 & & & \parallel \mathbf{next}(\ulcorner x : (g(v))\urcorner \parallel \ulcorner y : (v)\urcorner)).
\end{array}
$$

Definition $x\colon(z)$ 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 $g_{exch}(x, y)$ represents an exchange operation between the contents of x and y. If v is x's current value then $g(v)$ and v will be the next values of x and y respectively. In the case of functions that always return the same value (i.e. constants), we take the liberty of using that value as its symbol. For example, $\ulcorner x\colon(3)\urcorner \parallel \ulcorner y\colon(5)\urcorner \parallel \ulcorner 7_{exch}(x,y)\urcorner$ gives us the cells $x\colon(7)$ and $y\colon(3)$ in the next time interval. The assignment of v to a cell x, written $x \ := \ v$, can then be encoded as **local** y **in** $\ulcorner v_{\text{exch}}(x,y)\urcorner$ where the local variable y is used as dummy variable (cell).

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

Proposition 3.1 $\ulcorner x\colon(v)\urcorner \ \vdash \ (A \dot{\wedge} \dot{\neg}\, \mathtt{change}(x)) \dot{\Rightarrow} \circ A.$

[2] $[v_1/x_1, \ldots, v_m/x_m]$ is the operation of (syntactical) replacement of every occurrence of the x_i by v_i

3.2 The Zigzagging Example

An RCX is a programmable, controller-based LEGO® brick used to create autonomous robotic devices ([13]). Zigzagging [7] 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 $\mathtt{f},\mathtt{r},\mathtt{l} \in \mathit{Dom} - \{0\}$ and extend the signature with the predicate symbols `forward,right,left`.

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

Initially cells a_1 and a_2 contain neither $\mathtt{f}, r$ 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. The property below states that the robot indeed goes right and left infinitely often.

Proposition 3.2 $\ulcorner \mathit{GoZigzag}\urcorner \vdash \Box(\Diamond\mathtt{right} \mathbin{\dot\wedge} \Diamond\mathtt{left})$.

3.3 Multi-Agent Systems: The Pursuit Game Example

The Predator/Prey (or Pursuit) game [1] has been studied using a wide variety of approaches [11] and it has many different instantiations that can be used to illustrate different multi-agent scenarios [25]. As the Zigzagging example, instances of the Predator/Prey game have been modeled using autonomous robots [16]. 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 (with $max = 2^k$ 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

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

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 preys 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).

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

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

$$\begin{array}{llll} \mathit{Pred}_i \stackrel{\text{def}}{=} ! & (& \textbf{when}\ x_i = x & \textbf{do} \, (\ulcorner l := i \urcorner \parallel \ulcorner \mathit{Pursuit}_i \urcorner) \\ & + & \textbf{when}\ l = i \wedge x_{i \oplus 1} \neq x & \textbf{do} \ulcorner \mathit{Pursuit}_i \urcorner \\ & + & \textbf{when}\ l = i \oplus 1 \wedge x_i \neq x & \textbf{do} \ulcorner \mathit{Support}_i \urcorner \quad). \end{array}$$

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 $\mathit{Pursuit}_i$ defined below and keep doing it until the other predator $\mathit{Pred}_{i \oplus 1}$ declares itself the leader. If the other process is the leader then Pred_i runs process $\mathit{Support}_i$ defined below.

Process $\mathit{Pursuit}_i$, 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.

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

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

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

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 $p = max/2$.

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

The proposition states that the predators eventually capture the prey under our initial conditions.

Proposition 3.3 $Init \parallel Pred_0 \parallel Pred_1 \parallel Prey \vdash \Diamond(\mathtt{capture}_0 \dot{\vee} \mathtt{capture}_1)$.

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. $Init \parallel Pred_0 \parallel Prey \not\vdash \Diamond\mathtt{capture}_0$. A similar situation occurs if the predators were not allowed to alternate the leader role.

4 Behavioral Equivalence

In this section we introduce notions of equality for our calculus. We wish to distinguish between the observable behavior of two processes if the distinction can somehow be detected by a process interacting with them. A natural observation we can make of a process is its input-output behavior, i.e. its infinite sequences of input-output constraints.

Furthermore, in Section 2.3 we mentioned that we can model the behavior of processes in which each component is part of the environment of the others. Thus the only "external" input is the empty one, i.e., $\mathtt{true}^\omega$. Therefore, another interesting observation to make is the set of outputs on the empty sequence, which we shall call the language of a process.

We now introduce the observables and the corresponding equivalences we are interested in.

Definition 4.1 *Given P, the* input-output behavior *of P and the* language *of P are defined as*

$$io(P) = \{(\alpha, \alpha') \mid P \stackrel{(\alpha,\alpha')}{\Longrightarrow} {}^\omega\} \quad and \quad \mathcal{L}(P) = \{\alpha \mid P \stackrel{(\mathtt{true}^\omega,\alpha)}{\Longrightarrow} {}^\omega\},$$

respectively. For all P and Q, we define $P \sim_{io} Q$ iff $io(P) = io(Q)$ and $P \sim_{\mathcal{L}} Q$ iff $\mathcal{L}(P) = \mathcal{L}(Q)$.

Unfortunately, the equivalences $\sim_{io}$ and $\sim_{\mathcal{L}}$ are not preserved by process constructions, i.e. they are not *congruences*.

Example. Assume that a, b, c are non-equivalent constraints such that $c \vdash b \vdash a$. Let

$$\begin{array}{rcl} P & = & \mathbf{when}\ \mathtt{true}\ \mathbf{do}\ \mathbf{tell}(a)\ +\ \mathbf{when}\ (b)\ \mathbf{do}\ \mathbf{tell}(c) \\ Q & = & \mathbf{when}\ \mathtt{true}\ \mathbf{do}\ \mathbf{tell}(a)\ +\ \mathbf{when}\ (b)\ \mathbf{do}\ \mathbf{tell}(c) \\ & & + \\ & & \mathbf{when}\ \mathtt{true}\ \mathbf{do}\ (\mathbf{tell}(a) \parallel \mathbf{when}\ (b)\ \mathbf{do}\ \mathbf{tell}(c)) \end{array}$$

and let $R = \mathbf{when}\, a\, \mathbf{do}\, \mathbf{tell}(b)$. We leave it to the reader to verify that we can distinguish P from Q if we make R to interact with them, i.e. although $P \sim_{io} Q$ (and thus $P \sim_{\mathcal{L}} Q$) we have $R \parallel P \not\sim_{\mathcal{L}} R \parallel Q$ (and thus $R \parallel P \not\sim_{io} R \parallel Q$).

Therefore, we ought to consider the largest congruences included in $\sim_{io}$ and $\sim_{\mathcal{L}}$, respectively. More precisely,

Definition 4.2 *For all P and Q, $P \approx_{io} Q$ iff for every process context $C[.]$, $C[P] \sim_{io} C[Q]$, and $P \approx_{\mathcal{L}} Q$ iff for every process context $C[.]$, $C[P] \sim_{\mathcal{L}} C[Q]$.*

As usual a process context $C[.]$ is a process term with a single hole such that placing a process in the hole yields a well-formed process. The relations $\approx_{io}$ and $\approx_{\mathcal{L}}$ are then our first proper notion of equality for the calculus.

It is important to point out that the mismatch between $\approx_{io}$ and $\sim_{io}$ arises from allowing nondeterminism. In fact, the following result follows from ([18], Theorem 3).

Definition 4.3 *A process P is said to be* deterministic *iff for every construct of the form $\sum_{i\in I}$ **when** c_i **do** P_i in P, the c_i's are mutually exclusive.*

Proposition 4.4 *For all deterministic processes P and Q, $P \approx_{io} Q$ iff $P \sim_{io} Q$.*

The reason for using the name "deterministic process" is because given an input, the output of a process of this kind is always the same independently of the execution order of its parallel component [22].

Let us now see the relation between the different equivalences for arbitrary processes. The relation $\equiv$ denotes structural congruence (Definition 2.2). For technical purposes we consider the finite prefixes of the language of a process. Let $\mathcal{L}^i(P) = \{\alpha^i | \, \alpha \in \mathcal{L}(P)\}$ where α^i is the $i-th$ prefix of α and define $P \sim^i_{\mathcal{L}} Q$ iff $\mathcal{L}^i(P) = \mathcal{L}^i(Q)$. Obviously, relation $\sim_{\mathcal{L}}$ is weaker than $\sim_{io}$, however, the corresponding congruences coincide.

Theorem 4.5 $\equiv \, \subset \, \approx_{io} \, = \, \approx_{\mathcal{L}} \, \subset \, \sim_{io} \, \subset \, \sim_{\mathcal{L}} \, = \, \bigcap_{n\in\omega} \sim^n_{\mathcal{L}}$.

Proof: The proper inclusions are left for the reader to verify. The final equality follows from the fact that our calculus is finitely branching. Here we prove $\approx_{io} = \approx_{\mathcal{L}}$. The case $\approx_{io} \subseteq \approx_{\mathcal{L}}$ is trivial. We want to prove that $P \approx_{\mathcal{L}} Q$ implies $P \approx_{io} Q$. Suppose that $P \approx_{\mathcal{L}} Q$ but $P \not\approx_{io} Q$. Then there must exist a context $C[.]$ s.t $C[P] \not\sim_{io} C[Q]$. Consider the case $io(C[P]) \not\supset io(C[Q])$. Take an $\alpha = c_1.c_2 \ldots$ such that $(\alpha, \alpha') \in io(C[Q])$ but $(\alpha, \alpha') \notin io(C[P])$. There must then be a prefix of α' which differs from all other prefixes of sequences α'' s.t. $(\alpha, \alpha'') \in io(C[P])$. Suppose that this is the $n-$th prefix. One can verify that for the context

$$C'[.] = C[.] \;\parallel\; \prod_{i\leq n} \mathbf{next}^i \, \mathbf{tell}(c_i),$$

$\mathcal{L}(C'[P]) \neq \mathcal{L}(C'[Q])$. This contradicts our assumption $P \approx_{\mathcal{L}} Q$. The case $io(C[Q]) \not\supset io([P])$ is symmetric. Therefore $P \not\approx_{\mathcal{L}} Q$ as required. ■

We next investigate the type of contexts $C[.]$ in ntcc needed to verify $P \approx_{io} Q$ and focus on relation $\approx_{\mathcal{L}}$ as it is equivalent to $\approx_{io}$. The proposition below allows us to approximate the behavior of $!P$.

Proposition 4.6 *For all $P, Q, n \geq 0$: $Q \parallel !P \sim^n_{\mathcal{L}} Q \parallel \prod_{i\leq n} \mathbf{next}^i P$.*

The next proposition states that it is sufficient to consider parallel contexts.

Lemma 4.7 *$P \approx_{\mathcal{L}} Q$ iff for all R, $R \parallel P \sim_{\mathcal{L}} R \parallel Q$.*

Proof: Suppose that for all R, $P \parallel R \sim_{\mathcal{L}} Q \parallel R$. We can prove that for all contexts $C[.]$, $C[P] \parallel R \sim_{\mathcal{L}} C[Q] \parallel R$ for an arbitrary R. Here we outline the proof of the next and replication context cases. The other cases are trivial. For the next case we have $\mathbf{next}\, P \parallel R \xRightarrow{(c,c')} P \parallel R'$ iff $R \xRightarrow{(c,c')} R'$. Similarly, $\mathbf{next}\, Q \parallel R \xRightarrow{(c,c')} Q \parallel R'$ iff $R \xRightarrow{(c,c')} R'$. Thus, the result follows immediately from the initial assumption. As for the replication case, from the Prop. 4.6 for all n, $R \parallel !P \sim^n_{\mathcal{L}} R \parallel \prod_{i \leq n} \mathbf{next}^i P$ and $R \parallel !Q \sim^n_{\mathcal{L}} R \parallel \prod_{i \leq n} \mathbf{next}^i Q$. With the help of Theorem 4.5 ($\sim_{\mathcal{L}} = \bigcap_{n \in \omega} \sim^n_{\mathcal{L}}$) we get that $R \parallel !P \sim_{\mathcal{L}} R \parallel !Q$ if for all $n \geq 0$, $R \parallel !P \sim^n_{\mathcal{L}} R \parallel !Q$. The result now follows from the next and parallel cases. ∎

Moreover, if $\mathcal{C}$ (i.e., the underlying set of constraints) is finite we have the notion of a *universal context*, i.e., a context that can distinguish any two processes iff they are not language (or input-output) congruent. Intuitively, the idea is to provide a single process that can simulate all possible interactions that a process can have with others.

Consider $R \parallel P$ with P and R as in Example 4. By telling information, process P provides information which influences the evolution of R, i.e., the constraint a. Similarly, R influences the evolution of P by providing the constraint b. Thus asking a and then telling b is one possible interaction a process can have with P while telling a and then asking b is a possible interaction a process can have with R. In general, interactions can be represented as strictly increasing and alternating sequences of ask and tell operations (see [22]).

In the following we write $c' \prec c$ iff $c \vdash c'$ and $c \not\vdash c'$. The assertion $S \subseteq_{fin} S'$ holds iff S is a finite subset of S'. Given $S \subseteq_{fin} \mathcal{C}$, $ic(S)$ denotes the set of strictly increasing sequences in S^*, i.e., $ic(S) = \{c_1 \ldots c_n \in S^* \mid c_1 \prec c_2 \prec \ldots \prec c_n\}$. Furthermore, we extend the underlying constraint system signature Σ to a signature Σ' with unary predicates tr_β for each $\beta \in \mathcal{C}^*$. These predicates are "private" in the sense that they are only allowed to occur in the process contexts $U^S[.]$ defined below.

Definition 4.8 *The* distinguishing context *wrt* $S \subseteq_{fin} \mathcal{C}$, *written* $\mathcal{U}^S[.]$, *is defined as*

$$! \; (\sum_{\beta \in ic(S)} \mathbf{tell}(tr_\beta) \;\; \| \;\; \mathcal{T}_\beta) \parallel [.]$$

where for each $\beta \in S^*$, $\mathcal{T}_{c.\beta} = \mathbf{tell}(c) \parallel \mathcal{W}_\beta$ *and* $\mathcal{W}_{c.\beta} = \textbf{when } c \textbf{ do } \mathcal{T}_\beta$ *with* $\mathcal{T}_\epsilon = \mathcal{W}_\epsilon = \mathbf{skip}$.

Theorem 4.9 *Suppose that* $\mathcal{C}$ *is finite. Then* $P \approx_{\mathcal{L}} Q$ *iff* $\mathcal{U}^{\mathcal{C}}[P] \sim_{\mathcal{L}} \mathcal{U}^{\mathcal{C}}[Q]$.

Proof: The "only if" direction is trivial. Here we outline the proof of the "if" direction. From Lemma 4.7 it is sufficient to prove that $\mathcal{U}^{\mathcal{C}}[P] \sim_{\mathcal{L}} \mathcal{U}^{\mathcal{C}}[Q]$ implies $R \parallel P \sim_{\mathcal{L}} R \parallel Q$ for all R. Suppose that R is such that $R \parallel P \not\sim_{\mathcal{L}} R \parallel Q$. We want to prove that $\mathcal{U}^{\mathcal{C}}[P] \not\sim_{\mathcal{L}} \mathcal{U}^{\mathcal{C}}[Q]$.

Consider the case $\mathcal{L}(R \parallel P) \not\subseteq \mathcal{L}(R \parallel Q)$. Take an $\alpha = d_0.d_1 \ldots$ such that $\alpha \in \mathcal{L}(R \parallel P)$ and $\alpha \notin \mathcal{L}(R \parallel Q)$. Furthermore, suppose that $R_0 \parallel P_0 \stackrel{d_0}{\Longrightarrow} R_1 \parallel P_1 \stackrel{d_1}{\Longrightarrow} \ldots$ with $P = P_0$ and $R = R_0$.

We can represent the internal reduction of each $R_i \parallel P_i$ which gives us d_i and $R_{i+1} \parallel P_{i+1}$, as a sequence of internal transitions (or *interactions*) $\langle R_i^0 \parallel P_0^0, c_i^0 \rangle \longrightarrow^* \langle R_i^n \parallel P_i^n, c_i^n \rangle \not\longrightarrow$, with $R_i = R_i^0, P_i = P_i^0, c_i^0 = \texttt{true}, P_{i+1} = F(P_i^n), R_{i+1} = F(R_i^n)$ and $d_i = c_i^n$, satisfying

$$\begin{array}{llll}
\langle P_i^0, a_i^0 \rangle \longrightarrow^* & \langle P_i^1, a_i^1 \rangle & & \\
& \langle P_i^1, a_i^1 \wedge b_i^1 \rangle \longrightarrow^* & \langle P_i^2, a_i^2 \rangle & \\
& & \vdots & \\
& & \left\langle P_i^j, a_i^j \wedge b_i^j \right\rangle \longrightarrow^* & \left\langle P_i^{j+1}, a_i^{j+1} \right\rangle
\end{array}$$

$$\begin{array}{llll}
\langle R_i^0, b_i^0 \rangle \longrightarrow^* & \langle R_i^1, b_i^1 \rangle & & \\
& \langle R_i^1, a_i^1 \wedge b_i^1 \rangle \longrightarrow^* & \langle R_i^2, b_i^2 \rangle & \\
& & \vdots & \\
& & \left\langle R_i^j, a_i^j \wedge b_i^j \right\rangle \longrightarrow^* & \left\langle R_i^{j+1}, b_i^{j+1} \right\rangle
\end{array}$$

where for each $j \leq n$, $c_i^j = a_i^j \wedge b_i^j$. Let $\sigma_i = b_i^1.c_i^1 \ldots .b_i^n.c_i^n$. It is easy to see that $\langle \mathcal{T}_{\sigma_i} \parallel P_i^0, c_i^0 \rangle \longrightarrow^* \langle \mathcal{T}_\epsilon \parallel P_i^n, c_i^n \rangle \not\longrightarrow$ (see Definition 4.8). Note that sequence σ_i is increasing, thus by removing all constraint repetitions we get a strictly increasing sequence. Let β_i be such a sequence. One can verify that T_{β_i} can "mimic" R_i^0 interacting with P_i^0. More precisely,

$\langle \mathcal{T}_{\beta_i} \parallel P_i^0, c_i^0 \rangle \longrightarrow^* \langle \mathcal{T}_\epsilon \parallel P_i^n, c_i^n \rangle \not\longrightarrow$. This implies:

$$\left\langle !(\sum_{\beta \in ic(\mathcal{C})} \mathbf{tell}(tr_\beta) \ \parallel \ \mathcal{T}_\beta) \parallel P_i^0, \mathtt{true} \right\rangle \longrightarrow^* \langle \mathcal{T}_\epsilon \parallel P_i^n, d_i \wedge tr_{\beta_i} \rangle \not\longrightarrow \quad (1)$$

By observing that $last(\beta_i) = d_i$ (where $last(\beta_i)$ denotes the last element of β_i), one can show that R_i can mimic T_{β_i} interacting with any P' provided that the result is d_i. More precisely,:

$$\begin{array}{ll} \text{For all } P', & \text{if } \langle \mathcal{T}_{\beta_i} \parallel P', \mathtt{true} \rangle \longrightarrow^* \langle \mathcal{T}_\epsilon \parallel P'', d_i \rangle \not\longrightarrow \\ & \qquad \text{where } P' \longrightarrow^* P'', \\ & \text{then } \langle R_i^0 \parallel P', \mathtt{true} \rangle \longrightarrow^* \langle R_i^n \parallel P'', d_i \rangle \not\longrightarrow \qquad (2) \end{array}$$

¿From (1), $\alpha' = (d_0 \wedge tr_{\beta_0}).(d_1 \wedge tr_{\beta_1}) \ldots \in \mathcal{L}(!(\sum_{\beta \in ic(\mathcal{C})} \mathbf{tell}(tr_\beta) \parallel \mathcal{T}_\beta) \parallel P)$ where β_i corresponds to the internal $\mathcal{T}_{\beta_i}$ selected to "mimic" R_i. We want to show α' is not in $\mathcal{L}(!(\sum_{\beta \in ic(\mathcal{C})} \mathbf{tell}(tr_\beta) \ \parallel \ \mathcal{T}_\beta) \parallel Q)$. Suppose it is. Then at time i, T_{β_i} must be selected in the execution of $!(\sum_{\beta \in ic(\mathcal{C})} \mathbf{tell}(tr_\beta) \ \parallel \ T_\beta) \parallel Q$ that outputs α'. By using Property (2) (and observing our restriction on the use of tr_{β_i} predicates), one can inductively construct a sequence $R_0 \parallel Q_0 \stackrel{d_0}{\Longrightarrow} R_1 \parallel Q_1 \stackrel{d_1}{\Longrightarrow} \ldots$ with $Q = Q_0$, $R = R_0$. We conclude that $\alpha \in \mathcal{L}(R \parallel Q)$ thus contradicting the assumption about α.

The case of $\mathcal{L}(R \parallel Q) \not\subseteq \mathcal{L}(R \parallel P)$ is symmetric. ■

Therefore context $\mathcal{U}^{\mathcal{C}}[.]$ is the *universal* distinguishing context, provided that $\mathcal{C}$ is finite, as it can distinguish any two processes P and Q which are not language congruent.

It is interesting that even if $\mathcal{C}$ is not finite, we can construct specialized distinguishing contexts for arbitrary processes as stated in the following result. The idea is to choose a suitable finite set of constraints.

Definition 4.10 *Let $\Lambda \subset_{fin}$ Proc. Define $\mathcal{C}(\Lambda) \subseteq_{fin} \mathcal{C}$ as the set whose elements are* `true`*,* `false` *and all constraints resulting from the closure under conjunction and existential quantification of the constraints occurring in Λ's processes.*

Theorem 4.11 *For all $P, Q \in \Lambda \subset_{fin}$ Proc, $P \approx_{\mathcal{L}} Q$ iff $\mathcal{U}^{\mathcal{C}(\Lambda)}[P] \sim_{\mathcal{L}} \mathcal{U}^{\mathcal{C}(\Lambda)}[Q]$.*

Proof: The proof is the same as that of Theorem 4.9 except for the role of β_i which is now played by a sequence $\overline{\beta_i}$, defined below, that depends only on constraints in Λ's processes. More precisely, let $consq(c, S) = \{d \in S \mid c \vdash d\}$. Define $\overline{e}$ as the conjunction of all constraints in $consq(e, \mathcal{C}(\Lambda))$ and let $\overline{s}$ be the sequence that results from replacing each constraint e in a sequence s with $\overline{e}$. By definition every constraint in $\mathcal{C}(\Lambda)$ which can be inferred from e, can also be inferred from $\overline{e} \in \mathcal{C}(\Lambda)$. We proceed exactly as in the proof of Theorem 4.9 until properties (1) and (2), which we re-state as:

$$\left\langle !(\sum_{\beta \in ic(\mathcal{C}(\Lambda))} \mathbf{tell}(tr_\beta) \;\parallel\; \mathcal{T}_\beta) \parallel P_i^0, \mathtt{true} \right\rangle \longrightarrow^* \left\langle \mathcal{T}_\epsilon \parallel P_i^n, \overline{d_i} \wedge tr_{\overline{\beta_i}} \right\rangle \not\longrightarrow \tag{3}$$

and

$$\begin{aligned} \text{For all } P' \in \Lambda, \qquad & \text{if } \left\langle \mathcal{T}_{\overline{\beta_i}} \parallel P', \mathtt{true} \right\rangle \longrightarrow^* \left\langle \mathcal{T}_\epsilon \parallel P'', \overline{d_i} \right\rangle \not\longrightarrow \\ & \qquad\qquad \text{where } P' \longrightarrow^* P'', \\ & \text{then } \langle R_i^0 \parallel P', \mathtt{true} \rangle \longrightarrow^* \langle R_i^n \parallel P'', d_i \rangle \not\longrightarrow \end{aligned} \tag{4}$$

We then proceed as in the proof of Theorem 4.9; getting a contradiction out of (3) and (4). ■

Therefore $\mathcal{U}^{\mathcal{C}(\Lambda)}$ is an universal context for Λ's processes. The ability of constructing distinguishing contexts for arbitrary processes is important as it can be used for proving decidability results for $\approx_{io}$ (note that $P \approx_{\mathcal{L}} Q$ iff $\mathcal{U}^{\mathcal{C}(\{P,Q\})}[P] \sim_{\mathcal{L}} \mathcal{U}^{\mathcal{C}(\{P,Q\})}[Q]$). It turns out that $\sim_{\mathcal{L}}$ is decidable for a significant fragment of the calculus. The languages of these processes can be recognized by automata over infinite sequences, more precisely Büchi Automata ([3]). We will elaborate on this in the next section.

4.1 Decidability and Characterization of Processes Languages

In this section we will characterize processes languages in terms of ω-regular languages (i.e., the languages accepted by Büchi automata). Recall that Büchi automata are ordinary nondeterministic finite-state automata equipped with an acceptance condition that is appropriate for

ω-sequences: an ω-sequence is accepted if the automaton can read it from left to right while visiting a sequence of states in which some final state occurs infinitely often. This condition is called *Büchi acceptance* ([3]).

We aim at proving decidability of the relation $\sim_{\mathcal{L}}$ for a fragment of ntcc which we call *restricted-nondeterministic*.

Definition 4.12 *A process P is said to be* restricted-nondeterministic *iff for all* **local** x **in** Q *in* P*, for every construct of the form* $\sum_{i\in I}$ **when** c_i **do** Q_i *in* Q*, the* c_i*'s are mutually exclusive. We use* $Proc^r$ *to denote the set of all restricted-nondeterministic processes.*

This fragment allows non-deterministic process (summations) out of the scope of local variables. In fact, all application examples in this paper (Section 3) belong to this fragment. Notice that each **local** x **in** $P \in Proc^r$ is deterministic in the sense of Definition 4.3.

We shall show that the languages of restricted-nondeterministic processes are ω-regular. We will also show that given a $P \in Proc^r$ we can construct a Büchi automaton recognizing the language of P. Then using the fact that language equivalence for Büchi automata is decidable [23], we conclude that $\sim_{\mathcal{L}}$ is decidable for restricted-nondeterministic processes and thus so are $\approx_{\mathcal{L}}$ and $\approx_{io}$ (see Theorem 4.11).

To illustrate the problem in trying to use finite-state machines for representing processes let us consider the following example.

Example. Let $Q = \,!!P$ with $P = \sum_{j\in J} \mathbf{tell}(c_j)$. We have the following transition sequence (on input $\texttt{true}^\omega$):

$$Q \stackrel{d_1}{\Longrightarrow} Q \parallel !P \stackrel{d_2}{\Longrightarrow} Q \parallel !P \parallel !P \stackrel{d_3}{\Longrightarrow} \ldots \stackrel{d_n}{\Longrightarrow} Q \parallel \prod_n !P \stackrel{d_{n+1}}{\Longrightarrow} \ldots$$

This example illustrates that in a transition system where states are the elements of *Proc* it is possible to have infinite paths where all states are different up to structural congruence (i.e., there can be an infinite set of derivatives). Moreover, notice that in this particular example, the process at time i can output everything the process at time $i-1$ can, but not necessarily the other way round. This situation arises from the nondeterminism specified by P.

Nevertheless, we will show that after some time units the states can be identified up to $\approx_{\mathcal{L}}$. More precisely, the property we would like to have is that there exists t such that for all $k \geq t$, $\prod_k !P \approx_{\mathcal{L}} \prod_{k+1} !P$. In the above example for any $k \geq |J|$ we have $\prod_k !P \approx_{\mathcal{L}} \prod_{k+1} !P$ thus validating the property. Unfortunately, the property does not hold for processes out of $Proc^r$. Let us define an arbitrary-delay operation δP which delays P arbitrarily:

$$\delta P \stackrel{\text{def}}{=} P + \delta P.$$

The encoding in our calculus of the recursive definition of δP requires hiding over non-mutually exclusive summations (see [18]) thus it is out of $Proc^r$. Assume that $P = \mathbf{tell}(c)$. Then two copies of δP can output c at two (arbitrary) points of time while a single copy cannot. In general one can prove that for any $k > 1$, $\mathcal{L}(\prod_k \delta P) \subset \mathcal{L}(\prod_{k+1} \delta P)$, thus invalidating the property.

The following property is needed in the proof of Lemma 4.15 which implies the property described above. It relates the language of processes with the language of processes arising at intermediate steps of the internal computations.

Proposition 4.13 *$\alpha \in \mathcal{L}(P)$ iff there are Q and c such that $\langle P, \mathtt{true}\rangle \longrightarrow^* \langle Q, c\rangle$ and $Q \parallel \mathbf{tell}(c) \stackrel{\alpha}{\Longrightarrow}{}^{\omega}$.*

We now introduce the notion of multiplicity of a process.

Definition 4.14 *Let $m : Proc^r \rightarrow Nat$. The multiplicity of P, $m(P)$ is defined as*

$m(\mathbf{skip}) = 0$
$m(\mathbf{tell}(c)) = 1$
$m(\sum_{i\in I} \mathbf{when}\, c_i\, \mathbf{do}\, P_i) = \sum_{i \in I} m(P_i)$
$m(P \parallel Q) = max\{m(P), m(Q)\}$
$m(\mathbf{local}\, x\, \mathbf{in}\, P) = m(\mathbf{next} P) = m(\mathbf{unless}\, c\, \mathbf{next}\, P) = m(!P) = m(P)$.

The value $m(P)$ is aimed to be the number of copies of P, after which, further copies are redundant. This is stated in the following lemma which is the key for decidability of $\sim_{\mathcal{L}}$.

Lemma 4.15 *Let $P \in Proc^r$. For all $k > m(P)$, $\prod_{k-1} P \approx_{\mathcal{L}} \prod_k P$.*

Proof: The proof proceeds by induction on the structure of $P \in Proc^r$. Here we show some cases. Suppose $k > m(P)$.

• Case $P = P_1 \parallel P_2$. From Theorem 4.5 we get $\prod_k (P_1 \parallel P_2) \approx_{\mathcal{L}} \prod_k P_1 \parallel \prod_k P_2$. Note that $k > m(P) \geq m(P_1)$ and $k > m(P) \geq m(P_2)$. Therefore, from the hypothesis $\prod_k P_1 \parallel \prod_k P_2 \approx_{\mathcal{L}} \prod_{k-1} P_1 \parallel \prod_{k-1} P_2 \approx_{\mathcal{L}} \prod_{k-1}(P_1 \parallel P_2)$ as required.

• Case $P = \mathbf{next}\, Q$. We have $\prod_k \mathbf{next}\, Q \approx_{\mathcal{L}} \mathbf{next} \prod_k Q$ from Theorem 4.5. From $m(P) = m(Q)$, the hypothesis and Theorem 4.5, we get $\mathbf{next} \prod_k Q \approx_{\mathcal{L}} \mathbf{next} \prod_{k-1} Q \approx_{\mathcal{L}} \prod_{k-1} \mathbf{next}\, Q$.

• Case $P = !Q$. We verify that $\prod_k ! Q \approx_{\mathcal{L}} ! \prod_k Q$. From $m(P) = m(Q)$ and hypothesis we verify that $! \prod_k Q \approx_{\mathcal{L}} ! \prod_{k-1} Q \approx_{\mathcal{L}} \prod_{k-1} ! Q$.

• Case $P = \sum_{u \in I} \mathbf{when}\, c_u\, \mathbf{do}\, P_u$. ¿From Lemma 4.7 we know that it is enough to consider parallel contexts. Let E an arbitrary process and suppose that $\alpha = c.\alpha' \in \mathcal{L}(E \parallel \prod_k P)$ (1). We want to show that $\alpha \in \mathcal{L}(E \parallel \prod_{k-1} P)$. From (1) we know that there exists sequence of internal transitions $t = \langle E \parallel \prod_k P, \mathtt{true}\rangle \longrightarrow^* \gamma_1 \longrightarrow^*, ..., \longrightarrow^* \gamma_n \longrightarrow^* \langle R, c\rangle \not\longrightarrow$ with $\alpha' \in \mathcal{L}(F(R))$ which contains only the initial and final configuration, and those configurations $\gamma_1, ..., \gamma_n$ in which a reduction from a P takes place, if any. By monotonicity of the store if t contains a configuration with store c s.t. $\langle P, c\rangle \longrightarrow$ then since a reduction of each P must eventually take place $n = k$ ***(I)*** otherwise $n = 0$ (***II***).

(I). Suppose $n = k$. Define $E_0 = E$, $P_0 = \mathbf{skip}$. For $0 < j \leq n$, each γ_j can be defined as $\left\langle E_j \parallel P_j \parallel \prod_{n-j} P, c_j \right\rangle$, where $\langle E_{j-1} \parallel P_{j-1}, c_{j-1}\rangle \longrightarrow^* \left\langle E_j, c'_j \right\rangle$ for some c'_j s.t. $\left\langle P, c'_j \right\rangle \longrightarrow \langle P_j, c_j\rangle$ (a reduction from one of the k $P's$). Notice $k > m(P) = \Sigma_{Q: P \longrightarrow Q} m(Q)$, so from the pigeon-hole principle there must be a process P', $P \longrightarrow P'$ with $r > m(P')$ configurations $\gamma_{j_1}, \ldots \gamma_{j_r}$ such that each corresponding $P_{j_1}, \ldots, P_{j_r}$ is P'. Let γ_i be the first among these configurations and let P_i be the process in such a configuration, i.e., $E_i \parallel P' \parallel \prod_{k-i} P$. ¿From Proposition 4.13, we have $\alpha \in \mathcal{L}(P_i \parallel \mathbf{tell}(c_i))$. As r copies of P' are eventually triggered, one can verify that $\alpha \in \mathcal{L}(E_i \parallel \prod_r P' \parallel \prod_{k-(i+r-1)} P \parallel \mathbf{tell}(c_i))$. Since P' is a subprocess of P, from the hypothesis $\alpha \in \mathcal{L}(Q_i \parallel \mathbf{tell}(c_i))$ with $Q_i = E_i \parallel \prod_{r-1} P' \parallel \prod_{k-(i+r-1)} P$. One can then construct the sequence

$$\begin{aligned}
\left\langle E \parallel \textstyle\prod_{(k-1)} P, \texttt{true} \right\rangle &\longrightarrow^* \left\langle E_i \parallel P' \parallel \textstyle\prod_{(k-1)-i} P, c_i \right\rangle \\
&\longrightarrow \left\langle E_i \parallel \textstyle\prod_2 P' \parallel \textstyle\prod_{(k-1)-(i+1)} P, c_i \right\rangle \\
&\vdots \\
&\longrightarrow \left\langle E_i \parallel \textstyle\prod_{r-1} P' \parallel \textstyle\prod_{(k-1)-(i+r-2)} P, c_i \right\rangle \\
&= \langle Q_i, c_i \rangle .
\end{aligned}$$

From Proposition 4.13, $\alpha \in \mathcal{L}(E \parallel \prod_{(k-1)} P)$ as required.

(II). Suppose $n = 0$. Then $R = E' \parallel \prod_k P$ for some E' s.t. $\langle E, \texttt{true} \rangle \longrightarrow^* \langle E', c \rangle \not\longrightarrow$. Trivially $\left\langle E \parallel \prod_{k-1} P, \texttt{true} \right\rangle \longrightarrow \langle R', c \rangle \not\longrightarrow$ with $R' = E' \parallel \prod_{k-1} P$. From the definition of $F(.)$, $F(P) \equiv \mathbf{skip}$, thus $F(R) = F(E') \parallel \prod_k F(P) \equiv F(E') \equiv F(R') = F(E') \parallel \prod_{k-1} F(P)$. Hence $F(R) \approx_{\mathcal{L}} F(R')$ by Theorem 4.5, thus $\alpha' \in F(R')$. We then conclude $\alpha \in \mathcal{L}(E \parallel \prod_{k-1} P)$.

• Case $P = \mathbf{local}\ x\ \mathbf{in}\ Q$. In this case P is a deterministic process. It is easy to verify that if P is a deterministic process then $P \approx_{\mathcal{L}} \prod_k P$ for any k, thus validating the property. ■

The lemma below states that every language transition sequence over $Proc^r$ ultimately contains two language congruent processes.

Lemma 4.16 *Let $P_0 \stackrel{c_1}{\Longrightarrow} P_1 \stackrel{c_2}{\Longrightarrow} \ldots$ be an arbitrary language transition sequence where $P_0 \in Proc^r$. Then there are two processes P_m, P_n with $m < n$ such that $P_n \approx_{\mathcal{L}} P_m$.*

Proof: Let $P_0 \stackrel{c_1}{\Longrightarrow} P_1 \stackrel{c_2}{\Longrightarrow} \ldots$ be an arbitrary language transition sequence where $P_0 \in Proc^r$. It is sufficient to construct a sequence $P'_0 \stackrel{c_1}{\Longrightarrow}\approx_{\mathcal{L}} P'_1 \stackrel{c_2}{\Longrightarrow}\approx_{\mathcal{L}} \ldots$ with $P_i \approx_{\mathcal{L}} P'_i$ for every $i \geq 0$ and two processes P'_m, P'_n with $m < n$ satisfying $P'_n \equiv P'_m$ (Definition 2.2). We sketch such a construction next.

Every process P can be rewritten via $\equiv$ as $\prod_{i \in I} !R_i \parallel E$ where E is a replication-free processes. Hence $P_0 \stackrel{c_0}{\Longrightarrow} P_1 \stackrel{c_1}{\Longrightarrow} \ldots$ can be rewritten as:

$$\prod_{i \in I_0} !R_i \parallel E_0 \stackrel{c_0}{\Longrightarrow} \prod_{i \in I_1} !R_i \parallel E_1 \stackrel{c_1}{\Longrightarrow} \ldots \tag{5}$$

where each E_u is a non-replicated processes. It is easy to verify that $I_0 \subseteq I_1 \subseteq \ldots$ since new replicated processes can move up to the top level. Assume that k is such that satisfies $\prod_{i\in I_k} !R_i \approx_{\mathcal{L}} \prod_{i\in I_j} !R_i$ for any $j > k$. Such a k is guaranteed to exist from Lemma 4.15. Thus the sequence in (5) is point-wise $\approx_{\mathcal{L}}$-equivalent to the sequence

$$\prod_{i\in I_0} !R_i \parallel E_0 \xRightarrow{c_1} \ldots \prod_{i\in I_k} !R_i \parallel E_k \xRightarrow{c_k}\approx_{\mathcal{L}} \prod_{i\in I_k} !R_i \parallel E_{k+1} \xRightarrow{c_{k+1}}\approx_{\mathcal{L}} \ldots \tag{6}$$

Now notice that both the E_j's ($j > k$) and $\prod_{i\in I_k} R_i$ can have replicated processes $!R$ which can move up to the top level. However, $\prod_{i\in I_k} !R_i \parallel !R \approx_{\mathcal{L}} \prod_{i\in I_k} !R_i$ from our assumption about k. Therefore we can replace such replications with **skip**. Given Q let us use $\widehat{Q}$ to denote the processes resulting from replacing each replicated process in Q with **skip**. We can then verify that the sequence

$$\prod_{i\in I_0} !R_i \parallel E_0 \xRightarrow{c_1} \ldots \prod_{i\in I_k} !\widehat{R}_i \parallel \widehat{E}_k \xRightarrow{c_k}\approx_{\mathcal{L}} \prod_{i\in I_k} !\widehat{R}_i \parallel \widehat{E}_{k+1} \xRightarrow{c_{k+1}}\approx_{\mathcal{L}} \ldots \tag{7}$$

is point-wise $\approx_{\mathcal{L}}$-equivalent to the one in (6). We claim the following:

Claim 4.17 *For some $n > k$ there exists a m, with $k \leq m < n$ such that $\widehat{E}_m \equiv \widehat{E}_n$*

Thus, for m and n in the above claim, it follows $\prod_{i\in I_k} !\widehat{R}_i \parallel \widehat{E}_m \equiv \prod_{i\in I_k} !\widehat{R}_i \parallel \widehat{E}_n$ thus proving the Lemma. Below we prove this claim.

Define the *next-depth* of a process Q, written $nd(Q)$, as the maximum number of nesting of next operations in Q. Let $D(Q, i) = \{Q' \mid Q \Longrightarrow^i Q'\}$, i.e. the set of all processes which Q can possibly evolve to in i times units. Trivially, if Q is replication-free then for all $u > nd(Q)$, $D(Q, u) = \{\textbf{skip}\}$ (2).

Let $R = \prod_{i\in I_k} \widehat{R}_i$, $Rr = \prod_{i\in I_k} !\widehat{R}_i$ and $E = \widehat{E}_k$. Without loss of generality assume that $nd(R) > nd(E)$ (by adding next-guarded skips we can always augment the next-depth of a process). Let $h = nd(R)$. At time k the processes E and R are the ones to be executed in parallel with Rr. At time $k + 1$, a process in $D(E, 1)$, a process in $D(R, 1)$, and R which is a process in $D(R, 0)$ are the ones to be executed with Rr. In general, at time $k + n$ there are $n + 2$ processes $E' \in D(E, n)$, $Q_n \in D(R, n)$, $Q_{n-1} \in D(R, n-1), \ldots,$ $Q_0 \in D(R, 0)$ to be executed with Rr. If $n \geq h$, however, we know from (2) that at each following time unit it is enough to consider the process in the (finite) sets $D(R, 0), \ldots, D(R, h)$ since $D(R, u) = \{\textbf{skip}\}$ for $u > h$. The are $w = |D(R, 0)| \times \ldots \times |D(R, h)|$ many choices of the h process in these sets. Thus after $h + w$ time units at least one choice must be repeated. ■

By using the Lemma 4.16 we can prove that the set of derivatives of P, which we define as $S(P) = \{Q \mid P \stackrel{c_1}{\Longrightarrow} \dots \stackrel{c_n}{\Longrightarrow} Q\}$, modulo $\sim_{\mathcal{L}}$ is finite.

Lemma 4.18 *For every $P \in Proc^r$, $S(P)/\sim_{\mathcal{L}}$ is finite.*

Proof: Here we outline the proof. Consider the finitely-branching transition system graph of P with labeled transitions $\stackrel{c}{\Longrightarrow}$ modulo $\sim_{\mathcal{L}}$. One can verify that if the transition graph were infinite then it would have to have an infinite path $P \sim_{\mathcal{L}} Q_0 \stackrel{c_0}{\Longrightarrow}\sim_{\mathcal{L}} Q_1 \stackrel{c_1}{\Longrightarrow}\sim_{\mathcal{L}} Q_2 \dots$, where all the Q_i's are different (modulo $\sim_{\mathcal{L}}$). But this would imply that there is a sequence $P = P_0 \stackrel{c_0}{\Longrightarrow} P_1 \stackrel{c_1}{\Longrightarrow} P_2 \dots$ (with $P_i \sim_{\mathcal{L}} Q_i$ for all $i \geq 0$) where all the P_i's are different modulo $\approx_{\mathcal{L}}$ which is impossible according to Lemma 4.16 (Recall that from Theorem 4.5, $\approx_{\mathcal{L}} \subset \sim_{\mathcal{L}}$). ■

Given a restricted-nondeterministic process P, Lemma 4.18 above allows us to define a Büchi automaton $A_{P/\sim_{\mathcal{L}}}$ which accepts $\mathcal{L}(P)$. The set of states is $S(P)/\sim_{\mathcal{L}}$ in Lemma 4.18. All states are accepting. The start state is P. There is transition from Q to Q' labeled by c iff $Q \stackrel{c}{\Longrightarrow} Q'$. It is easy to verify such an automaton accepts $\mathcal{L}(P)$.

Theorem 4.19 *For every $P \in Proc^r$, $\mathcal{L}(P)$ is an ω-regular language.*

The definition of $A_{P/\sim_{\mathcal{L}}}$ above does not give us an effective way of constructing the automaton. In Algorithm 1 we describe a method which given $P \in Proc^r$ constructs a Büchi automaton A_P accepting $\mathcal{L}(P)$.

First we need the following definitions: given Q and R let $r(R, Q)$ be the number of occurrences of R in Q at the top level. Let $Q \downarrow_R$ the process that results from replacing with **skip** each non-top-level occurrence of $!R$ in Q if $r(!R, Q) > m(!R)$ (See Definition 4.14). Let $Q \uparrow_R$ be the process that results from replacing with **skip**, $r(!R, Q) - m(!R)$ top-level occurrence of $!R$ in Q in some fixed order. Suppose that we enumerate all the replicated process in Q in some fixed order $R_1, \dots, R_n$. Let us define $Q\downarrow$ as the process $Q \downarrow_{R_1} \dots \downarrow_{R_n}$ and $Q \uparrow$ as $Q \uparrow_{R_1} \dots \uparrow_{R_n}$. Recall that $\equiv$ denotes the structural congruence (Definition 2.2).

Remark 4.20 *For each permutation π on $\{1, \dots, m\}$,*
$Q \uparrow_{R_1} \dots \uparrow_{R_m} \equiv Q \uparrow_{R_{\pi(1)}} \dots \uparrow_{R_{\pi(m)}}$ *and* $Q \downarrow_{R_1} \dots \downarrow_{R_m} \equiv Q \downarrow_{R_{\pi(1)}} \dots \downarrow_{R_{\pi(m)}}$

The proposition below follows from Lemma 4.15.

Proposition 4.21 *For all Q, $Q\downarrow \approx_{\mathcal{L}} Q\uparrow \approx_{\mathcal{L}} Q$.*

Algorithm 1 Constructing the automaton A_P

---

1. Start by creating the initial state and label it with $(P\downarrow\uparrow)$.

2. Choose a state p' labeled by P' from the current transition graph and a reduction $P' \stackrel{c}{\Longrightarrow} Q$. The choice should satisfy that there is not already an edge labeled with c from p' to a state q with a label structurally congruent to $(Q\downarrow\uparrow)$. If such a choice is not possible we stop.

3. If there is already a state q labeled with a process (structurally equivalent to) $(Q\downarrow\uparrow)$ then create an edge from p' to it with label c. Otherwise create a new state q with label $(Q\downarrow\uparrow)$ and an edge from p' to it with label c.

4. Go to (2).

This algorithm assumes decidability of $\equiv$ which basically follows from the decidability of the π-calculus structural congruence *without* the replication axiom [15]. The termination of Algorithm 1 is based on the proof of Lemma 4.16. Basically, each path in the transition graph constructed by this method is constructed as in the proof of the lemma; if the method did not terminate then the construction would violate the claim in the proof. The partial correctness is easy to verify.

Theorem 4.22 *For all $P \in Proc^r$, one can effectively construct a Büchi automaton A_P accepting the set $\mathcal{L}(P)$.*

Therefore $\sim_{\mathcal{L}}$ is decidable for restricted-nondeterministic processes (Definition 4.1). Moreover, $\approx_{\mathcal{L}} = \approx_{io}$ is also decidable for these processes as we need to consider only one context to check whether two processes are language congruent (Theorem 4.11).

Corollary 4.23 *Relations $\sim_{\mathcal{L}}$, $\approx_{\mathcal{L}}$ and $\approx_{io}$ are decidable for restricted nondeterministic processes.*

5 Automata for the Strongest-Postcondition and Input-Output Behavior

In the previous sections we introduced the notion of the language of a process and described how to construct automata for the language of restricted-nondeterministic processes. In this section we use such notions and constructions and relate them to the strongest postcondition and input-output behavior of processes.

Henceforth we assume that the underlying set of constraints $\mathcal{C}$ is the set of all constraints in $FD[max]$ (Section 3). The following results, however, can be extended to $\mathcal{C}$'s defined over finite-domain constraint systems or, of course, to finite $\mathcal{C}$'s. Notice that $\mathcal{C}$ is infinite since we have infinitely many variables

Given P, we want to find a finite set which contains the "relevant" inputs for a processes P. We shall prove in Lemma 5.1 that if x does not occur in the free-variables of P, $fv(P)$, then the reductions of P on $c[x]$ can be identified with those of P on $\exists_x c[x]$. We can then say that if a constraint has all its free variables in $fv(P)$ is "relevant" for P, otherwise is "irrelevant".

The next lemma states that indeed we can abstract away from the variables not occurring free in the program.

Lemma 5.1 *For all $x \notin fv(P)$, $P \overset{(c[x],d[x])}{\Longrightarrow} Q$ if and only if $P \overset{(\exists_x c, \exists_x d)}{\Longrightarrow} Q$*

Proof: We first prove the following property: For all P, Q and $x \notin fv(P)$, $\langle P, c[x]\rangle \longrightarrow \langle Q, d[x]\rangle$ iff $\langle P, \exists_x c\rangle \longrightarrow \langle Q, \exists_x d\rangle$ (1).

Both directions proceeed by induction on (depth of the) inference. Here we only show the "only if" direction as the other case is simlar. Suppose that $x \notin fv(P)$. Consider the final step of the inference of $\langle P, c[x]\rangle \longrightarrow \langle Q, d[x]\rangle$. These are the key cases.

- Suppose that it is inferred by using rule TELL where $P = \mathbf{tell}(e)$, $Q = \mathbf{skip}$ and $d = c \wedge e$. Since $x \notin fv(e)$, $\exists_x c \wedge e = \exists_x(c \wedge e)$. Then, by using TELL $\langle P, \exists_x c\rangle \longrightarrow \langle Q, \exists_x d\rangle$.
- Suppose that it is inferred by using rule ASK where $P = \sum_{i \in I} \mathbf{when}\ (c_i)\ \mathbf{do} P_i$, $Q = P_j$ for $j \in J$ s.t. $c \vdash c_j$, and $d = c$. Since $x \notin fv(c_j)$, one can verify that $c \vdash c_j$ iff $\exists_x c \vdash c_j$. Thus, by using ASK $\langle P, \exists_x c\rangle \longrightarrow \langle Q, \exists_x d\rangle$.
- Suppose that it is inferred by using rule PAR $P = P_1 \parallel P_2$ and $Q = P_1' \parallel P_2$ with $\langle P_1, c\rangle \longrightarrow \langle P_1', d\rangle$ inferred by a shorter inference. We have

$x \notin fv(P_1)$ thus by appeal to induction we get $\langle P_1, \exists_x c\rangle \longrightarrow \langle P_1', \exists_x d\rangle$. Then, by using PAR we obtain $\langle P, \exists_x c\rangle \longrightarrow \langle Q, \exists_x d\rangle$ as required.
• Suppose that it is inferred by using rule LOC where for some variable ν P = **local** (ν, e) **in** P', Q = **local** (ν, e') **in** Q', $d = c \wedge \exists_\nu e'$ with $\langle P', e \wedge \exists_\nu c\rangle \longrightarrow \langle Q, e'\rangle$ inferred by a shorter inference. First, suppose that ν is x. In this case by using the shorter inference in the premise of LOC and simple logic manipulations we have:

$$\frac{\langle P', e \wedge \exists_x\exists_x c\rangle = \langle P', e \wedge \exists_x c\rangle \ \longrightarrow\ \langle Q', e'\rangle}{\begin{array}{rl}\langle \mathbf{local}\ (x,e)\ \mathbf{in}\ P',\ \exists_x c\ \rangle \ \longrightarrow & \langle \mathbf{local}\ (x,e')\ \mathbf{in}\ Q', \exists_x c \wedge \exists_x e'\rangle \\ = & \langle \mathbf{local}\ (x,e')\ \mathbf{in}\ Q', \exists_x (c \wedge \exists_x e')\rangle\end{array}}$$

as wanted. Now suppose that v is not x. In this case $x \notin fv(P')$ and $x \notin fv(e)$. So we can α-convert P by replacing ν with x without risk of capture. By using STRUCT after LOC we get the desired result.

It follows from (1) above and a simple induction on n that for $x \notin fv(P)$, $\langle P, c[x]\rangle \longrightarrow^n \langle Q, d[x]\rangle$ iff $\langle P, \exists_x c\rangle \longrightarrow^n \langle Q, \exists_x d\rangle$ which proves the Lemma. □ ■

In what follows $\delta_V c$ where V is set of variables, denotes the constraint resulting from the existential quantification of each free variable in c not in V (e.g. $\delta_{\{x,y\}}(x + w = z) = \exists_w\exists_z(x + w = z)$). We extend this notation to other structures; $\delta_V t$, $\delta_V(\alpha)$ and $\delta_V(S)$ denotes the element-wise application of δ_P to a tuple t, sequence α and set S, respectively. For simplicity, we will write $\delta_{P_1,\ldots,P_n}$ instead of $\delta_{fv(P_1)\cup\ldots\cup fv(P_n)}$. Thus, each constraint $\delta_{P_1,\ldots,P_n} c$ is of the form $\exists_{x_1\ldots x_m} c$ where none of the x_i's a free-variable of some P_j. The following corollary follows inmediately from Lemma 5.1.

Corollary 5.2 *$P \stackrel{(c,d)}{\Longrightarrow} Q$ if and only if $P \stackrel{\delta_P(c,d)}{\Longrightarrow} Q$*

Notice that $\delta_P(\mathcal{C}) = \{c \in \mathcal{C} \mid fv(c) \subseteq\ fv(P)\ \}$, i.e., the set of what we called "relevant" constraints for P. Moreover, there are only finitely constraints in $\delta_P(\mathcal{C})$ since the set of free-variables of a P is finite and $\mathcal{C}$ is defined over the finite-domain constraint system $FD[max]$.

Proposition 5.3 *The set $\delta_P(\mathcal{C})$ is finite.*

Therefore from Corollary 5.2 it follows that the finite set $\delta_P(\mathcal{C})^\omega$ is the contains all the relevant inputs for P.

Let us define $\mathcal{I}_S\ =\ !\ (\sum_{c\in S} \mathbf{tell}(c))$. Notice that in $R = P \parallel \mathcal{I}_S$, process $\mathcal{I}_S$ can provide P with all inputs in S^ω. More precisely, the set of outputs of P on inputs in S^ω is the same as the set of outputs of R on the empty input $\texttt{true}^\omega$, i.e. the language of R. Hence in $P \parallel \mathcal{I}_{\delta_P}$ process $\mathcal{I}_{\delta_P}$ can provide all sequences of relevant input constraints for P. Now, recall that the strongest postcondition of P, $sp(P)$ (Definition 2.4) is the set all sequences that P can possibly output on some input, i.e, the set of outputs of P on inputs in $\mathcal{C}$. This leads to the next result wich can be proven with the help of Corollary 5.2.

Lemma 5.4 *For any P, $\mathcal{L}(P \parallel \mathcal{I}_{\delta_P}) = \delta_P(sp(P)) \subseteq sp(P)$.*

In Section 2.4 we define the assertion $P \models B$, where B is temporal formula, to hold iff the strongest-postcondition of P, $sp(P) \subseteq [\![B]\!]$, where $\subseteq [\![B]\!]$ is the set of models of B). We confine ourselves to assertions $P \models B$ in which B refers only to variables of the program P, i.e., $fv(B) \subseteq fv(P)$. The following theorem relates $P \models B$ to the language of $P \parallel \mathcal{I}_{\delta_P}$.

Theorem 5.5 *For all P and temporal formula B with $fv(B) \subseteq fv(P)$*

$$P \models B \quad \textit{iff} \quad \mathcal{L}(P \ \parallel \ \mathcal{O}_{\mathcal{I}_\mathcal{P}} \subseteq [\![B]\!]$$

Proof: The "only if" direction follows from Lemma 5.4 Suppose that $\mathcal{L}(P \ \parallel \ \mathcal{I}_{\delta_P}) \subseteq [\![B]\!]$. From Lemma 5.4 $\delta_P\alpha \models B$ for all $\alpha \in sp(P)$. Whence, for all $\alpha \in sp(P)$, $\alpha \models B$ since each assertion $\delta_P\alpha \models B$ holds independently of the variables outside of $fv(P)$. $\Box$ $\blacksquare$

We now proceed to relate input-output equivalence $\sim_{io}$ (Definition 4.1) with language equivalence of certain kind of Büchi automata derived from the previous constructions. Let $P \in Proc^r$ and $S \subseteq_{fin} \mathcal{C}$. We show next how to construct a Büchi automaton A_P^S over the alphabet $(S \times \mathcal{C})^\omega$ for the input-ouput behavior of P on inputs in S^ω. The automaton is constructed from the automaton $A_{P\parallel\mathcal{I}_S}$ which recognizes the language of $P \parallel \mathcal{I}_S$. Notice that states of $A_{P\parallel\mathcal{I}_S}$ are labelled by processes of the form $Q \parallel \mathcal{I}_S$ for some Q. The construction is described in Algorithm 2.

Remark 5.6 *Consider an arbitrary infinite path from the starting state in the transition graph of A_P^S whose arcs are labelled succesively $(c_1, c_1').(c_2, c_2')\ldots$. The sequence $(c_1, c_1').(c_2, c_2')\ldots$ is accepted by A_P^S since $c_1'.c_2'\ldots$ is accepted by $A_{P\parallel\mathcal{I}_S}$ and the two paths go through the same states. It should also be clear that $(c_1.c_2.\ldots\ ,\ c_1'.c_2'\ldots) \in io(P)$.*

Algorithm 2 Constructing the automaton A_P^S

1. Start by making the start state, the accepting states, and the set of states of $A_{P\|\mathcal{I}_S}$ to be the corresponding ones for A_P^S.

2. Take a transition in $A_{P\|\mathcal{I}_S}$ labelled by d from a state q labeled with $Q \parallel \mathcal{I}_S$ to a state r labelled $R \parallel \mathcal{I}_S$.

3. Then create a new transition from q labelled with Q to r labelled with R. Make (c, d) the label of the transition, where $c \in S$, if $Q \xRightarrow{(c,d)} R'$ with $R' \approx_{\mathcal{L}} R$.

4. Add the transition to the current transition graph of A_P^S.

5. Repeat (2-4) until no new transition can be added.

¿From the construction of A_P^S and Corollary 5.2 we can prove the following result relating the input-output behaviour of a process P and the language of $A_P^{\delta_P(\mathcal{C})}$.

Proposition 5.7 *For every restricted-nondeterministic process P*

$$(c_1.c_2.\dots\ ,\ c_1'.c_2'\dots) \in io(P) \quad \textit{iff} \quad \delta_P((c_1, c_1').(c_2.c_2')\dots) \in \mathcal{L}(A_P^{\delta_P(\mathcal{C})}).$$

We can also reduce the question of input-output equivalence between processes to language equivalence in Büchi automata as stated in the theorem below.

Theorem 5.8 *For all $P, Q \in Proc^r$, $P \sim_{io} Q$ iff $\mathcal{L}(A_P^{\delta_{P,Q}\mathcal{C}}) = \mathcal{L}(A_Q^{\delta_{P,Q}\mathcal{C}})$*

Proof: The "only if" direction is trivial. Suppose that $\mathcal{L}(A_P^{\delta_{P,Q}\mathcal{C}}) = \mathcal{L}(A_Q^{\delta_{P,Q}\mathcal{C}})$ (1). Consider the case $io(P) \subseteq io(Q)$. Take an arbitrary $(\alpha, \alpha') \in io(P)$ with $\alpha = c_1.c_2.\dots$ and $\alpha' = c_1'.c_2'.\dots$. From the construction of $A_P^{\delta_{P,Q}\mathcal{C}}$ (and with the help of Corollary 5.2) we can verify that $\delta_{P,Q}((c_1.c_1').(c_2.c_2').\dots) \in \mathcal{L}(A_P^{\delta_{P,Q}\mathcal{C}})$. Hence from our assumption $\delta_{P,Q}((c_1.c_1').(c_2.c_2').\dots) \in \mathcal{L}(A_Q^{\delta_{P,Q}\mathcal{C}})$. This implies (Remark 5.6) that $\delta_{P,Q}(\alpha, \alpha') \in Q$. Notice that the constraints $\delta_{P,Q}c_i$ and $\delta_{P,Q}c_i'$ are of the form $\exists_{x_1\dots x_n} c_i$ and $\exists_{x_1\dots x_n} c_i'$, resp., where none of the x_i's is in $fv(Q)$. By using Corollary 5.2 one verifies that $(\alpha, \alpha') \in Q$. The case $io(Q) \subseteq io(P)$

is symmetric. $\Box$ ■

We can also use the automaton $A_P^{\mathcal{C}[P]}$ as a simple input-output execution model for P. We conclude this section by defining such a model of execution.

Input-Output Execution of $A_P^{\mathcal{C}[P]}$: The execution starts at time unit 0 in the initial state of the automaton. At time i the automaton is in some state p. Given an input c_i from the environment the automaton nondeterministically selects a transition labeled by $(\delta_{fv(P)}c_i, \delta_{fv(P)}d_i)$ from p to some q. The constraint $d'_i = c_i \wedge \delta_{fv(P)}d_i$ is the output to the environment at time i. The automaton then moves to state q and repeat this process (at time $i+1$). We then say that on input $\alpha = c_1.c_2\ldots$ the automaton *can produce* an output $\alpha' = d'_1.d'_2.\ldots$. Let us denote by $io(A_P^{io})$ the set of all (α, α') such that on input α, A_P^{io} can produce α'.

The correspondence between the execution of processes and the execution of its automaton is stated by the following theorem.

Theorem 5.9 *For all $P \in Proc^r$, $io(P) = io(A_P^{io})$.*

Proof: Here we prove $io(P) \subseteq io(A_P^{io})$ as the other direction is trivial. Suppose that $(\alpha, \alpha') \in io(P)$, with $\alpha = c_1.c_2\ldots$ and $\alpha = c'_1.c_2.\ldots$. By monotonicity of the store each $c'_i = c_i \wedge d_i$ with d_i s.t. $fv(d_i) \subseteq fv(P)$. From proposition 5.7 we know that $\delta_P((c_1, c'_1), (c_2, c'_2), \ldots) \in \mathcal{L}(A_P^{io})$. On each each c_i the automaton can output $c_i \wedge \delta_P c'_i$. Notice that each $\delta_P c'_i$ must then have the form $(\exists_{x_1,\ldots,x_m} c_i) \wedge d_i$. So, $c_i \wedge \delta_P c'_i = c_i \wedge d_i = c'_i$. Consequentely, on α, the automaton A_P^{io} can output α'. $\Box$. ■

6 Related Work and Concluding Remarks

6.0.1 Related Work.

The work most closely related to our paper is that of tcc ([20]). Our proposal is a strict extension of tcc, in the sense that tcc can be encoded in (the deterministic fragment of) ntcc, while the vice-versa is not possible because tcc does not have constructs to express non-determinism. The input-output behavior of tcc has been studied in [20]. In tcc the input-output equivalence and congruence coincide as only deterministic processes are allowed. Therefore, there is no need for the study of universal or distinguishing contexts as in the ntcc case. In [20] it was shown that tcc processes can be compiled into (deterministic) finite-state automata. Moreover such a compilation is compositional. This result relies on determinacy of tcc processes. As shown in this paper, in ntcc the non-deterministic constructs are the ones which present technical difficulties to deal with when trying to represent them as finite-state machines. Other interesting extensions of tcc have been proposed in [9, 10, 21]. None of these, however, consider non-determinism.

The tccp calculus ([5]) is the only other proposal for a non-deterministic timed extension of ccp that we know of. As such, tccp provides a declarative language for the specification of (large) timed systems. 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. Like ntcc, the deterministic fragment of tccp can be used to program reactive systems. A store that grows monotonically, however, may be inadequate for the kind of application we have in mind, like RCX micro-controllers.

A proof system for reasoning about the correctness of tccp processes was recently introduced in [4]. The underlying temporal logic in [4] can be used for describing input-output behavior while the one in [17] for ntcc 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 [14]. This may come in handy when using the Consequence Rule present in the proof systems of both [4] and [17].

6.0.2 Concluding Remarks.

In this paper we introduced and studied different notions of equality for ntcc. We showed that the languages of restricted-nondeterministic processes can be characterized in terms of ω-languages. Furthermore, we described how to construct Büchi automata accepting the language of restricted-nondeterministic processes. This allowed us to prove decidability of language-equivalence for these processes. By proving the existence of distinguishing contexts, and that the input-output and language congruences coincide, we also proved decidability for these relations. We have also used the automata constructions for characterizing the strongest postcondition and input-output behavior of processes. This gives us some decidability results for these notions and also a simple execution model for restricted-nondeterministic processes.

On the practical side we show applications examples illustrating the expressiveness of (the restricted-nondeterministic fragment of) ntcc.

Our current research includes the study of the decidability of $\approx_{\mathcal{L}}$ for arbitrary ntcc processes as it remains an open question. The plan for future research includes the extension of ntcc to a probabilistic model following ideas in [12] and [8]. This is justified by the existence of RCX program examples involving stochastic behavior which cannot be faithfully modeled with non-deterministic behavior. In a more practical setting we plan to define a programming language for RCX controllers based on ntcc.

Acknowledgments.

We owe much to Catuscia Palamidessi, with whom we have worked on ntcc, for her insight into ccp issues. We would also like to thank Maurizio Gabbrielli, Paulo Oliva, Pawel Sobocinski, Jesus Almansa for helpful comments on different aspects of ntcc.

References

[1] M. Benda, V. Jagannathan, and R. Dodhiawala. On optimal cooperation of knowledge sources - an empirical investigation. Technical Report BCS-G2010-28, Boeing Advanced Technology Center, 1986.

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

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

[4] F. de Boer, M. Gabbrielli, and M. Chiara. A temporal logic for reasoning about timed concurrent constraint programs. In *TIME 01*. IEEE Press, 2001.

[5] F. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Information and Computation*, 1999. To appear.

[6] F. S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 19(5):685–725, 1997.

[7] J. Fredslund. The assumption architecture. Progress Report, Department of Computer Science, University of Aarhus, November 1999.

[8] V. Gupta, R. Jagadeesan, and P. Panangaden. Stochastic processes as concurrent constraint programs. In *Symposium on Principles of Programming Languages*, pages 189–202, 1999.

[9] V. Gupta, R. Jagadeesan, and V. Saraswat. Models for concurrent constraint programming. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 66–83, 26–29 August 1996.

[10] V. Gupta, R. Jagadeesan, and V. Saraswat. Probabilistic concurrent constraint programming. In *CONCUR '97: Concurrency Theory, 8th International Conference*, volume 1243 of *LNCS*, pages 243–257, 1–4 July 1997.

[11] T. Haynes and S. Sen. The evolution of multiagent coordination strategies. *Adaptive Behavior*, 1997.

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

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

[14] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification*. Springer, 1991.

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

[16] S. Nolfi and D. Floreano. Coevolving predator and prey robots: Do "arms races" arise in artificial evolution? *Artificial Life*, 4(4):311–335, 1998.

[17] C. Palamidessi and F. Valencia. A temporal concurrent constraint programming calculus. In *Proc. of the Seventh International Conference on Principles and Practice of Constraint Programming*, 26 November 2001.

[18] C. Palamidessi and F. Valencia. A temporal constraint programming calculus. Technical Report RS-01-20, BRICS, University of Aarhus, June 2001. availabe via http://www.brics.dk/~fvalenci/publications.html.

[19] V. Saraswat. *Concurrent Constraint Programming*. The MIT Press, Cambridge, MA, 1993.

[20] V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, 4–7 July 1994.

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

[22] V. Saraswat, M. Rinard, and P. Panangaden. The semantic foundations of concurrent constraint programming. In *POPL '91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages*, pages 333–352, 21–23 January 1991.

[23] A. Sistla, M. Vardi, and P. Wolper. The complementation problem for buchi automata with applications to temporal logic. *Theoretical Computer Science*, 49:217–237, 1987.

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

[25] P. Stone and M. Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8:345–383, 2000.

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

Table 1: An operational semantics for ntcc. The upper part defines the internal transitions while the lower part defines the observable transitions. The function F, used in OBS, is given in Definition 2.3

Recent BRICS Report Series Publications

RS-01-48 Mogens Nielsen and Frank D. Valencia. *Temporal Concurrent Constraint Programming: Applications and Behavior*. December 2001. 36 pp.

RS-01-47 Jesper Buus Nielsen. *Non-Committing Encryption is Too Easy in the Random Oracle Model*. December 2001. 20 pp.

RS-01-46 Lars Kristiansen. *The Implicit Computational Complexity of Imperative Programming Languages*. November 2001. 46 pp.

RS-01-45 Ivan B. Damgård and Gudmund Skovbjerg Frandsen. *An Extended Quadratic Frobenius Primality Test with Average Case Error Estimates*. November 2001. 43 pp.

RS-01-44 M. Oliver Möller, Harald Rueß, and Maria Sorea. *Predicate Abstraction for Dense Real-Time Systems*. November 2001. 27 pp.

RS-01-43 Ivan B. Damgård and Jesper Buus Nielsen. *From Known-Plaintext Security to Chosen-Plaintext Security*. November 2001. 18 pp.

RS-01-42 Zoltán Ésik and Werner Kuich. *Rationally Additive Semirings*. November 2001. 11 pp.

RS-01-41 Ivan B. Damgård and Jesper Buus Nielsen. *Perfect Hiding and Perfect Binding Universally Composable Commitment Schemes with Constant Expansion Factor*. October 2001. 43 pp.

RS-01-40 Daniel Damian and Olivier Danvy. *CPS Transformation of Flow Information, Part II: Administrative Reductions*. October 2001. 9 pp.

RS-01-39 Olivier Danvy and Mayer Goldberg. *There and Back Again*. October 2001. 14 pp.

RS-01-38 Zoltán Ésik. *Free De Morgan Bisemigroups and Bisemilattices*. October 2001. 13 pp.

RS-01-37 Ronald Cramer and Victor Shoup. *Universal Hash Proofs and a Paradigm for Adaptive Chosen Ciphertext Secure Public-Key Encryption*. October 2001. 34 pp.

RS-01-36 Gerth Stølting Brodal, Rolf Fagerberg, and Riko Jacob. *Cache Oblivious Search Trees via Binary Trees of Small Height*. October 2001. 20 pp.