Improving the Real-Time Concurrent Constraint Calculus with a Delay Declaration

Gerardo M. Sarria M. *

Abstract—**The Real-Time Concurrent Constraint Programming Calculus (rtcc) is a model of concurrency developed to specify systems with real-time behaviour. In this paper we enhance this calculus by extending the concept of time as a discrete sequence of minimal units that we will call *ticks*. We also add a new construct to rtcc to be able of delaying the execution of a process for an amount of ticks. The operational semantics were adapted to support these new features. We argue that this extension makes the calculus temporally homogeneous and allows modeling real-time systems (such as an improvisation system where time is an inflexible notion) in a more precise way.**

Keywords: process calculi, rtcc, operational semantics, delay declaration

1 Introduction

The rtcc calculus [11] is a ccp-based formalism [10], extension of the ntcc calculus [8]. rtcc is obtained from ntcc by adding a construct for specifying strong preemption and by extending the transition system with support for resources, limited time and true concurrency. This calculus allows modeling real-time and reactive behaviour.

In reactive systems, time is conceptually divided into *discrete intervals* (or *time units*). In a time interval, a process receives a stimulus from the environment, it computes (reacts) and responds to the environment. In the case of rtcc the stimulus is a tuple consisting of a constraint representing the initial store, the available number of resources and the duration of the time unit, and responds with another tuple consisting of a constraint representing the final store, the maximum number of resources used in calculations and the time spent in them. A reactive system is shown in figure 1. For each P_i there is an stimulus $\langle d_i, r_i, t_i\rangle$ and a response $\langle d_i', r_i', t_i'\rangle$ in the time unit k_i.

To model real time, we assume that each time unit is a clock-cycle in which computations (internal transitions) involving addition of information to the store (*tell* operations) and querying the store (*ask* operations) take a particular amount of time dependent on the constraint system. A discrete global clock is introduced and it is assumed that this clock is synchronized with the physical time (i.e. two successive time units in this calculus correspond exactly to two moments in the physical time). We also assume that the environment provides the exact duration of the time unit. That is, processes may not have all the time they need to run, instead, if they do not reach their resting point in a particular time, some (or all) of their computations not done will be discarded before the time unit is over. The duration will be then the available time that processes have to execute. We will take this available time as a natural number; this allows to think of time as a discrete sequence of minimal units that we will call *ticks*.

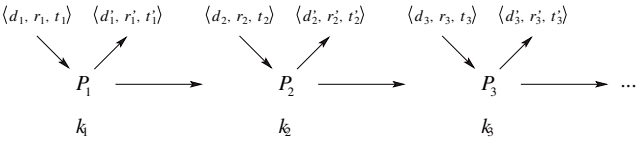

Figure 1: Reactive System

The rtcc calculus provides a way of executing unit delays and weak time-outs with the constructs **next** P and **unless** c **next** P. We realized that just with these constructs a calculus is not able to express neither strong time-outs [1]: "if an event A does not happen by time t, cause event B to happen at time t", nor real delays within the current time unit.

Process **next** P activates P the next time unit. Then this construct delays a process an amount of time given by the environment (the duration of the time unit). This means that there is no total control over the exact duration of the retard and might be more than the time wanted. To eliminate this drawback, we will add the construct:

delay P **for** δ

It will delay the execution of process P for at least δ ticks. This process allows to express things like "this process should start 3 seconds after another starts". This construct is similar to the notation of delay declarations introduced in logic languages in [7], and used in programming languages like Gödel [4].

*AVISPA Research Group. Pontificia Universidad Javeriana, Cali - Colombia. Email: gsarria@cic.javerianacali.edu.co

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

The main contributions of this paper are: 1) the introduction to `rtcc` of a new construct to delay the execution of a process within a time unit, 2) an example illustrating the potential of the new feature, 3) an extension of the operational semantics to support the new construct, and 4) the explanation of some properties of processes.

2 The Calculus

Here we describe the enhanced syntax and the extended operational semantics for `rtcc`. We begin by introducing the notion of constraint system, very important in ccp-based calculi.

Constraint System. The `rtcc` processes are parameterized in a *constraint system* which specifies what kind of constraints handle the model. Formally, it is a pair (Σ, Δ) where Σ is a signature (a set of constants, functions and predicates) and Δ is a first order theory over Σ (a set of first-order sentences with at least one model).

Given a constraint system, the underlying language $\mathcal{L}$ of the constraint system is a tuple $(\Sigma, \mathcal{V}, \mathcal{S})$, where $\mathcal{V}$ is a set of variables, and $\mathcal{S}$ is a set with the symbols $\neg, \wedge, \vee, \Rightarrow, \exists, \forall$ and the predicates `true` and `false`. A *constraint* is a first-order formulae constructed in $\mathcal{L}$.

A constraint c *entails* a constraint d in Δ, notation $c \vDash_\Delta d$, iff $c \Rightarrow d$ is true in all models of Δ. The entailment relation is written $\vDash$ instead of $\vDash_\Delta$ if Δ can be inferred from the context.

For a constraint system D, the set of elements of the constraint system is denoted by $|D|$ and $|D|_0$ represents its set of finite elements. The set of constraints in the underlying constraint system will be denoted by $\mathcal{C}$. The conjunction of all posted constraints will be called *the store*.

Process Syntax. The 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:

$$\begin{array}{rcl} P, Q, \ldots & ::= & \mathbf{tell}(c) \mid \sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i \mid P \parallel Q \\ & \mid & \mathbf{local}\ x\ \mathbf{in}\ P \mid \mathbf{unless}\ c\ \mathbf{next}\ P \\ & \mid & \mathbf{catch}\ c\ \mathbf{in}\ P\ \mathbf{finally}\ Q \mid \mathbf{next}\ P \\ & \mid & \mathbf{delay}\ P\ \mathbf{for}\ \delta \mid\ !P \mid \star P \end{array}$$

Intuitively, the process $\mathbf{tell}(c)$ adds constraint c to the store within the current time unit. The ask process **when** c **do** P is generalized with a non-deterministic choice of the form $\sum_{i\in I}$ **when** c_i **do** P_i (I is a finite set of indices). This process, in the current time unit, must non-deterministically choose one of the P_j ($j \in I$) whose corresponding guard constraint c_j is entailed by the store, and execute it. The non-chosen processes are precluded. Two processes P and Q acting concurrently are denoted by the process $P \parallel Q$. In one time unit P and Q operate in parallel, communicating through the store by telling and asking information. The "$\parallel$" operator is defined as left associative. The process **local** x **in** P declares a variable x private to P (hidden to other processes). This process behaves like P, except that all information about x produced by P can only be seen by P and the information about x produced by other processes is hidden to P. The weak time-out process, **unless** c **next** P, represents the activation of P the next time unit if c cannot be inferred from the store in the current time interval (i.e. $d \nvDash c$). Otherwise, P will be discarded. The strong time-out process, **catch** c **in** P **finally** Q, represents the interruption of P in the current time interval when the store can entail c; otherwise, the execution of P continues. When process P is interrupted, process Q is executed. If P finishes, Q is discarded.

The execution of a process P now can be delayed in two ways: with **delay** P **for** δ the process P is activated in the current time unit but at least δ ticks after the beginning of the time unit, whilst with **next** P the process P will be activated in the next time interval. The operator "**!**" is used to define infinite behaviour. The process $!P$ represents $P \parallel$ **next** $P \parallel$ **next(next** P**)** $\parallel \ldots$, (i.e. $!P$ executes P in the current time unit and it is replicated in the next time interval). An arbitrary (but finite) delay is represented with the operator "$\star$". The process $\star P$ represents an unbounded but finite P + **next** P + **next(next** P**)** + ..., (i.e. it allows to model asynchronous behaviour across the time intervals).

The guarded-choice summation process $\sum_{i\in I}$ **when** c_i **do** P_i is actually the abbreviation of

$$\mathbf{when}\ c_{i_1}\ \mathbf{do}\ P_{i_1} + \ldots + \mathbf{when}\ c_{i_n}\ \mathbf{do}\ P_{i_n}$$

where $I = \{i_1, \ldots, i_n\}$. The symbol "+" is used for binary summations (similar to the choice operator from CCS [6]). If there is no ambiguities, the "**when** c **do**" can be omitted when $c =$ `true`, that is, $\sum_{i\in I} P_i$. The process that do nothing is **skip**. The inactivity process is defined as the empty summation $\sum_{i\in\emptyset} P_i$. This process is similar to process 0 of CCS and $STOP$ of CSP [5]. Furthermore, terminated processes will always behave like **skip**. We write $\prod_{i\in I} P_i$, where $I = \{i_1, \ldots, i_n\}$ to denote the parallel composition of all the P_i, that is, $P_{i_1} \parallel \ldots \parallel P_{i_n}$. When process Q is **skip**, the "**finally** Q" part in process **catch** c **in** P **finally** Q can be omitted, that is, we can write **catch** c **in** P. A nest of delta delay processes such as **delay** (**delay** P **for** δ_1) **for** δ_2 can be abbreviated to **delay** P **for** $\delta_1 + \delta_2$. Notation **next**^{n} P (where **next** is repeated n times) is written to abbreviate the process **next** (**next** (... (**next** P) ...)). A bounded replication and asynchrony can be specified using summation and product. $!_I P$ and $\star_I P$ are defined as abbreviations for

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

$\prod_{i\in I}\mathbf{next}^i P$ and $\sum_{i\in I}\mathbf{next}^i P$, respectively. For example, process $!_{[m,n]}P$ means that P is always active between the next m and $m+n$ time units.

Now we will show a simple example illustrating the specification of temporal behaviour in this calculus.

Example 2.1. Suppose a simple improvisation situation where there are two machines M_1 and M_2. The first machine M_1 performs a single random action from a list *Actions* every 15 ticks. The second machine M_2 must follow it, that is, perform a series of actions depending on the action performed by M_1. Additionally, in some occasions M_1 not only performs a single action but two in the same time unit (it performs one action and 5 ticks latter performs another). In this case M_2 must stop its performance and try to follow the second action (there may be cases in which this is not possible due to the limit of time). This behaviour can be modeled as follows:

First, we have to model M_1:

$$M_1 \stackrel{\text{def}}{=} \mathbf{!}\sum_{i\in Actions}\mathbf{tell}(action1 = i) \parallel \star\ \mathbf{delay}\sum_{i\in Actions}\mathbf{tell}(action2 = i)\ \mathbf{for}\ 5$$

Now for the second machine we assume a process *FollowingActions* that calculates the actions to follow and performs them. Also, we assume an action $0 \notin Actions$. Thus M_2 is modeled:

$$M_2 \stackrel{\text{def}}{=} \mathbf{!}\ \mathbf{when}\ action1 \neq 0\ \mathbf{do}\ \mathbf{catch}\ action2 \neq 0\ \mathbf{in}\ FollowingActions_{(action1)}\ \mathbf{finally}\ FollowingActions_{(action2)}$$

To model the whole system we simply launch the process $M_1 \parallel M_2$. □

3 Operational Semantics

The operational semantics can be formally described by means of a transition system conformed by the set of processes *Proc*, the set of configurations Γ and transition relations $\rightarrow$ and $\Rightarrow$. A configuration γ is a tuple $\langle P, d, t\rangle$ where P is a process, d is a constraint in $\mathcal{C}$ representing the store, and t is the amount of time left to the process to be executed. The transition relations $\rightarrow = \{\stackrel{\langle r\rangle}{\longrightarrow}, r \in \mathbb{Z}^+\}$ and $\Rightarrow$ are the least relations satisfying the rules in tables 1 and 2.

The *internal* transition rule $\langle P, d, t\rangle \stackrel{r}{\rightarrow} \langle P', d', t'\rangle$ means that in one internal time using r resources process P with store d and available time t reduces to process P' with store d' and leaves t' time remaining. We write $\langle P, d, t\rangle \rightarrow \langle P', d', t'\rangle$ (omitting the "r") when resources

Table 1: Internal Transition Rules of `rtcc`

$$\frac{t - \Phi_T(c,d) \geq 0}{\langle \mathbf{tell}(c), d, t\rangle \stackrel{1}{\rightarrow} \langle \mathbf{skip}, d \wedge c, t - \Phi_T(c,d)\rangle}$$

$$\frac{t - \Phi_A(c_j,d) \geq 0 \qquad d \vDash c_j, \qquad j \in I}{\langle \sum_{i\in I}\mathbf{when}\ c_i\ \mathbf{do}\ P_i, d, t\rangle \stackrel{1}{\rightarrow} \langle P_j, d, t - \Phi_A(c_j,d)\rangle}$$

$$\frac{\langle P,d,t\rangle \stackrel{s_p}{\longrightarrow} \langle P', d'_p, t'_p\rangle \quad \langle Q,d,t\rangle \stackrel{s_q}{\longrightarrow} \langle Q', d'_q, t'_q\rangle \quad s_p + s_q \leq r}{\langle P \parallel Q, d, t\rangle \stackrel{s_p+s_q}{\longrightarrow} \langle P' \parallel Q', d'_p \wedge d'_q, \min(t'_p, t'_q)\rangle}$$

$$\frac{\langle P,d,t\rangle \stackrel{s_p}{\longrightarrow} \langle P', d'_p, t'_p\rangle \quad s_p \leq r}{\langle P \parallel Q, d, t\rangle \stackrel{s_p}{\longrightarrow} \langle P' \parallel Q, d'_p, t'_p\rangle}$$

$$\frac{\langle Q,d,t\rangle \stackrel{s_q}{\longrightarrow} \langle Q', d'_q, t'_q\rangle \quad s_q \leq r}{\langle P \parallel Q, d, t\rangle \stackrel{s_q}{\longrightarrow} \langle P \parallel Q', d'_q, t'_q\rangle}$$

$$\frac{\langle P, c \wedge \exists_x d, t - \Phi_T(c, \exists_x d)\rangle \stackrel{s}{\rightarrow} \langle P', c', t'\rangle}{\langle \mathbf{local}\ x, c\ \mathbf{in}\ P, d, t\rangle \stackrel{s}{\rightarrow} \langle \mathbf{local}\ x, c'\ \mathbf{in}\ P', d \wedge \exists_x c', t'\rangle}$$

$$\frac{t - \Phi_A(c,d) \geq 0 \qquad d \vDash c}{\langle \mathbf{unless}\ c\ \mathbf{next}\ P, d, t\rangle \stackrel{1}{\rightarrow} \langle \mathbf{skip}, d, t - \Phi_A(c,d)\rangle}$$

$$\frac{t - \Phi_A(c,d) \geq 0 \qquad d \vDash c}{\langle \mathbf{catch}\ c\ \mathbf{in}\ P\ \mathbf{finally}\ Q, d, t\rangle \stackrel{1}{\rightarrow} \langle Q, d, t - \Phi_A(c,d)\rangle}$$

$$\frac{\langle P, d, t - \Phi_A(c,d)\rangle \stackrel{s}{\rightarrow} \langle P', d', t'\rangle \qquad d \nvDash c}{\langle \mathbf{catch}\ c\ \mathbf{in}\ P\ \mathbf{finally}\ Q, d, t\rangle \stackrel{s}{\rightarrow} \langle \mathbf{catch}\ c\ \mathbf{in}\ P'\ \mathbf{finally}\ Q, d', t'\rangle}$$

$$\frac{\delta > T - t \qquad t > 0}{\langle \mathbf{delay}\ P\ \mathbf{for}\ \delta, d, t\rangle \stackrel{0}{\rightarrow} \langle \mathbf{delay}\ P\ \mathbf{for}\ \delta, d, t-1\rangle}$$

$$\frac{\delta \leq T - t}{\langle \mathbf{delay}\ P\ \mathbf{for}\ \delta, d, t\rangle \stackrel{0}{\rightarrow} \langle P, d, t\rangle}$$

$$\frac{}{\langle !P, d, t\rangle \stackrel{0}{\rightarrow} \langle P \parallel \mathbf{next}\ !P, d, t\rangle}$$

$$\frac{}{\langle \star P, d, t\rangle \stackrel{0}{\rightarrow} \langle \mathbf{next}^m\ P, d, t\rangle}\ \text{if } m \geq 0$$

$$\frac{\gamma_1 \rightarrow \gamma_2}{\gamma'_1 \rightarrow \gamma'_2}\ \text{if } \gamma_1 \equiv \gamma'_1 \text{ and } \gamma_2 \equiv \gamma'_2$$

Table 2: Observable Transition Rule of `rtcc`

$$\frac{\langle P, c, t\rangle \rightarrow_S^* \langle Q, d, t'\rangle \nrightarrow}{P \stackrel{(\langle c, r, t\rangle,\ \langle d, \max(S), t - t'\rangle)}{\Longrightarrow} R}\ \text{if } R \equiv F(Q)$$

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

are not relevant. The *observable* transition rule $P \stackrel{(\iota,o)}{\Longrightarrow} Q$ means that process P given an input ι from the environment reduces to process Q and outputs o to the environment in one time unit. Input ι is a tuple consisting of the initial store c, the number of resources available r within the time unit and the duration t of the time unit. Output o is also a tuple consisting of the resulting store d, the maximum number of resources r' used by processes and the time spent t' by all process to be executed. An observable transition is constructed from a sequence of internal transitions. It is assumed that internal transitions cannot be directly observed.

Now we are going to explain the transitions rules in tables 1 and 2. A tell process adds a constraint to the current store and terminates, unless there is not enough time to execute it (in this case it remains blocked). The time left to other processes after evolving is equal to the time available before the transition less the time spent by the constraint system to add the constraint to the store. The time spent by the constraint system is given by functions $\Phi_T, \Phi_A : |D|_0 \times |D|_0 \longrightarrow \mathbb{N} - \{0\}$ ($\Phi_T(c,d)$ approximates the time spent in adding constraint c to store d, and $\Phi_A(c,d)$ estimates the time querying if the store d can entail a constraint c). In addition, execution of a tell operation requires one resource.

The rule for a choice says that the process chooses one of the processes whose corresponding guard is entailed by the store and execute it, unless it has not enough time to query the store in which case it remains blocked. Computation of the time left is as for the tell process. The store in this operation is not modified. It consumes one resource unit.

The first rule of parallel composition says that both processes P and Q executes concurrently if the amount of resources needed by both processes separately is less than or equal to the number of resources available. The resulting store is the conjunction of the output stores from the execution of both processes separately. This process terminates iff both processes do. Therefore, the time left is the minimum of those times left by each process. The second and third rules affirm that in a parallel process, only one of the two processes can evolve due to the number of resources available.

To define the rule for locality, following [3], we extend the construct of local behaviour to **local** x, c **in** P to represent the evolution of the process. Variable c is the local information (or store) produced during the evolution. Initially, c is empty, so we regard **local** x **in** P as **local** x,`true` **in** P. The rule for locality says that if P can evolve to P' with a store composed by c and information of the "global" store d not involving x (variable x in d is hidden to P), then the **local ... in** P process reduces to a **local ... in** P' process where d is enlarged with information about the resulting local store c' without the information on x (x in c' is hidden to d and, therefore, to external processes).

In a weak time-out process, if c is entailed by the store, process P is terminated. Otherwise it will behave like **next** P. This will be explained below with the rule for observations. For a strong time-out, a process P ends its execution (and another process Q starts) if a constraint c is entailed by the store. Otherwise it evolves but asking for entailment of constraint persists.

The two rules for delaying state that a process **delay** P **for** δ delays the execution of P for at least δ ticks. Once the delay is less than the current internal time (T represents the duration of the time-unit given by the environment), the process reduces to P (i.e. it will be activated). In each transition this process does not consume any resource.

The replication rule specifies that the process P will be executed in the current time unit and then copy itself (process $!P$) to the next time unit. The rule for asynchrony says that a process P will be delayed for an unbounded but finite time, that is, P will be executed some time in the future (but not in the past). The rule that allows to use the structural congruence relation $\equiv$ defined below states that structurally congruent configurations have the same reductions.

Finally, the rule for observable transitions states that a process P evolves to R in one time unit if there is a sequence of internal transitions starting in configuration $\langle P, c, t\rangle$ and ending in configuration $\langle Q, d, t'\rangle$. Process R, called the "residual process", is constituted by the processes to be executed in the next time unit. The latter are obtained from Q by applying the future function defined as follows:

Let $F : Proc \to Proc$ be defined by

$$F(Q) = \begin{cases} R & \text{if } Q = \textbf{next } R \text{ or} \\ & \quad Q = \textbf{unless } c \textbf{ next } R \\ F(Q_1) \parallel F(Q_2) & \text{if } Q = Q_1 \parallel Q_2 \\ \textbf{catch } c \textbf{ in } F(R) \textbf{ finally } S & \text{if } Q = \textbf{catch } c \textbf{ in } R \textbf{ finally } S \\ \textbf{local } x \textbf{ in } F(R) & \text{if } Q = \textbf{local } x, c \textbf{ in } R \\ \textbf{skip} & \text{Otherwise} \end{cases}$$

To simplify the transitions, a congruence relation $\equiv$ is defined. Following [10], we introduce the standard notions of contexts and behavioural equivalence.

Informally, a context is a phrase (an expression) with a single hole, denoted by $[\cdot]$, that can be plugged in with processes. Formally, processes context C is defined by the following syntax:

$$\begin{array}{lcll} C & ::= & [\cdot] & \mid \textbf{when } c \textbf{ do } C + M \\ & \mid & C \parallel C & \mid \textbf{local } x \textbf{ in } C \\ & \mid & \textbf{unless } c \textbf{ next } C & \mid \textbf{catch } c \textbf{ in } C \textbf{ finally } C \\ & \mid & \textbf{delay } C \textbf{ for } \delta & \mid \textbf{next } C \\ & \mid & \textbf{!} C & \mid \star C \end{array}$$

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

where M stands for summations.

Two processes P and Q are *equivalent*, notation $P \doteq Q$, if for any context C, $P \doteq Q$ implies $C[P] \doteq C[Q]$. Let $\equiv$ be the smallest equivalence relation over processes satisfying:

1. $P \equiv Q$ if they only differ by a renaming of bound variables
2. $P \parallel \mathbf{skip} \equiv \mathbf{skip} \parallel P \equiv P$
3. $P \parallel Q \equiv Q \parallel P$
4. $\mathbf{next\ skip} \equiv \mathbf{skip}$
5. $\mathbf{local}\ x\ \mathbf{in\ skip} \equiv \mathbf{skip}$
6. $\mathbf{local}\ x\ y\ \mathbf{in}\ P \equiv \mathbf{local}\ y\ x\ \mathbf{in}\ P$
7. $\mathbf{local}\ x\ \mathbf{in\ next}\ P \equiv \mathbf{next}(\mathbf{local}\ x\ \mathbf{in}\ P)$

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

Properties. It is clear that with the introduction of the strong time-out construct, the delta delay construct and the additional observables of the transition system not all ccp properties hold. For example, the properties of monotonicity with respect to the store (if a process P evolve to Q given a particular store d, then P also evolves to Q given a stronger store e, $e \vDash d$) and restartability explained in [3] do not hold since for a given store a process may evolve, but if that particular store is augmented, it is possible that the signal that stops the process (with the **catch** construct) be now present, so the process evolves in a different way. Moreover, time becomes very important because processes are limited by the available time. This available time is reduced in every transition, so if we take the output of a process and we give it to the same process as input, that process might evolve in another way obtaining different results. This show that the notion of quiescent point, usual in CCP calculi, involves time now.

The following two properties state that a process can only post constraints in the store or leave it unmodified, but cannot take out constraints from it, i.e. the store can only be augmented, not reduced. Additionally, a process consumes some time to evolve, that is, the time available at the beginning of the transition is always greater than or equal to the time at the end (since processes ultimately perform ask and tell operations, they reduce the available time using functions Φ_A and Φ_T, in other words, available time in a transition is always reducing.

Property 3.1. (Internal Extensiveness). *If* $\langle P, c, t\rangle \to \langle Q, d, t'\rangle$ *then* $d \vDash c$ *and* $t > t' \geq 0$.

Proof. The proof proceeds by simple induction on the inference of $\langle P, c, t\rangle \to \langle Q, d, t'\rangle$. □

The property above can be extended to the observable relation.

Property 3.2. (Observable Extensiveness). *If* $P \xRightarrow{(\langle c,r,t\rangle,\ \langle d,s,t'\rangle)} Q$ *then* $d \vDash c$ *and* $t > t' \geq 0$.

Proof. By definition, if $P \xRightarrow{(\langle c,r,t\rangle,\ \langle d,s,t'\rangle)} Q$, then there is a sequence

$$\langle P_1, c_1, t_1\rangle \to \langle P_2, c_2, t_2\rangle \to \ldots \to \langle P_n, c_n, t_n\rangle \nrightarrow$$

with $P = P_1$, $Q = F(P_n)$, $c = c_1$, $t = t_1$, $d = c_n$ and $t' = t - t_n$. Then, by property 3.1 $c_n \vDash \ldots \vDash c_2 \vDash c_1$ and $t_1 > \ldots > t_n \geq 0$. Hence $d \vDash c$ and $t > t' \geq 0$. □

Time introduces a different behaviour of transitions than that of `ntcc`. For example, suppose that there is 5 ticks of available time and we have two processes executing in parallel $P_1 \stackrel{\text{def}}{=} \mathbf{tell}(x = 0)$ and $P_2 \stackrel{\text{def}}{=} \mathbf{catch}\ x = 0\ \mathbf{in}\ Q_1\ \mathbf{finally}\ Q_2$. If the current store is not strong enough to infer $x = 0$ and posting that constraint in the store takes 6 ticks of time, P_1 cannot add it so process Q_1 will continue its execution; but if we augment the amount of available time the constraint will be added, Q_1 will be stopped and Q_2 probably will be executed (if there's time). We can find a similar situations with other constructs.

Note that resources were not considered in the above properties. This can be explained with the fact that processes can evolve with just a single resource, they would only need enough time.

Finally, since each time unit has a fixed time given by the environment, the number of internal transitions is finite, i.e. there is always a final transition in a sequence. This is important since it guarantees that there are no infinite computations in one time unit.

Theorem 3.3. *Every sequence of internal transitions is finite.*

Proof. The proof follows directly from the fact that $\forall c, d \in |D|_0,\ \ \Phi_T(c,d) > 0$ and $\Phi_A(c,d) > 0$, and from property 3.1. □

4 Concluding Remarks

In this paper we enhanced the real-time concurrent constraint calculus `rtcc` with a delay construct. We believe that now with this new feature this calculus allows to model real-time behaviour in a more precise way. We extended the operational semantics with two new rules

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

allowing to express the internal transitions involving delaying a process within a time unit.

We also showed the applicability of the new features by modeling an improvisation system. Previously in [9] we showed the musical expressiveness of the `rtcc` calculus by modeling musical dissonances.

The new construct **delay** P **for** δ arose from the catch process for two purposes: (1) given the transition system proposed where two processes can be executed at the same time (true concurrency), if there is no way to delay the execution of a process within a time unit, every process would be executed simultaneously (assuming that there are enough resources) (2) it makes the calculus homogeneous with respect to the notion of time, that is, now we can delay a process for some given ticks or for some given time units.

A delay declaration similar to **delay** P **for** δ was first introduced in a `ccp`-based language in [2] (it was called δ-CCP). However the concept of delay in that model is different from our approach. In the δ-CCP calculus, the delay mechanism is simulated by modifying the ask construct: the agent $ask(\delta(\overline{x})) \rightarrow A$ behaves like A if the current store satisfies the property $\delta(\overline{x})$ (a user-defined predicate), otherwise the agent suspends.

5 Acknowledgments

We want to thank Camilo Rueda for his brilliant ideas and support during this research. We also thank Salim Perchy for studying `rtcc` and letting us know some early problems involving the delays.

References

[1] Berry, G.: Preemption in concurrent systems. In: Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science. pp. 72–93. Springer-Verlag, London, UK (1993)

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

[3] de Boer, F.S., Pierro, A.D., Palamidessi, C.: Nondeterminism and infinite computations in constraint programming. In: Selected Papers of the Workshop on Topology and Completion in Semantics. Theoretical Computer Science, vol. 151, pp. 37–78. Elsevier Science Publishers B. V., Chartres, France (1995)

[4] Hill, P., Lloyd, J.: The Gödel Programming Language. The MIT Press (April 1994)

[5] Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International Series in Computer Science, Prentice Hall (April 1985)

[6] Milner, R.: A Calculus of Communicating Systems. Lecture Notes in Computer Science, Springer-Verlag (1980)

[7] Naish, L.: An introduction to mu-prolog. Tech. Rep. 82/2, The University of Melbourne, Melbourne, Australia (1982)

[8] Palamidessi, C., Valencia, F.: A temporal concurrent constraint programming calculus. In: Seventh International Conference on Principles and Practice of Constraint Programming. Lecture Notes in Computer Science, vol. 2239, pp. 302–316. Springer-Verlang, London, UK (December 2001)

[9] Perchy, S., Sarria, G.: Dissonances: Brief description and its computational representation in the rtcc calculus. In: 6th Sound and Music Computing Conference (SMC2009). Porto, Portugal (July 2009)

[10] Saraswat, V.A.: Concurrent Constraint Programming. ACM Doctoral Dissertation Award, The MIT Press, Cambridge, MA, USA (1993)

[11] Sarria, G., Rueda, C.: Real-time concurrent constraint programming. In: 34th Latinamerican Conference on Informatics (CLEI2008). Santa Fe, Argentina (September 2008)

ISBN: 978-988-17012-0-6
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)