Real-Time Concurrent Constraint Calculus: The Complete Operational Semantics

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 provide the complete operational semantics of this calculus.**

***Index Terms*—process calculi, rtcc, operational semantics**

I. Introduction

The rtcc calculus [1], [2] is a ccp-based formalism [3], extension of the ntcc calculus [4]. rtcc is obtained from ntcc by adding constructs for specifying strong preemption and delay declarations, 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. A reactive system is shown in figure 1.

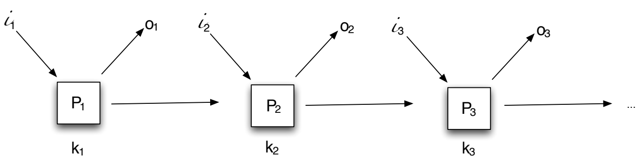

Fig. 1. Reactive System

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*.

Manuscript received January 12, 2011; revised January 14, 2011.

G. Sarria is with the Department of Science and Engineering of Computing, Pontificia Universidad Javeriana, Cali - Colombia, and is a member of the AVISPA Research Group. Email: gsarria@cic.javerianacali.edu.co

Now, most formal models of processes abstract away many properties of real systems such as duration of actions and number of processors [5]. Others assume maximal parallelism, that is the assumption of having n processors to execute n parallel processes (as in [6]). Nevertheless, for real-time systems the fact that processes have to share one processor cannot be ignored, since it may influence both the temporal and the functional behaviour of the system [7]. Moreover, as it is said in [8] the temporal behaviour of a real-time system depends not only on delays due to process synchronization, but also on the availability of shared resources. In this sense, we assume that the environment also provides a number r of resources. Each process P takes some of these. When P is finished, it releases them.

Then in the case of rtcc the stimulus ι_i provided by the environment of the reactive system is a tuple consisting of a constraint representing the initial store, the available number of resources and the duration of the time unit, and the response o_i of the process is another tuple consisting of a constraint representing the final store, the maximum number of resources used in calculations and the time spent in them. Formally, we can say that 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.

II. The Calculus

Here we describe the syntax and the 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).

An example of a widely used constraint system is the Finite-Domain Constraint System **FD**$[max]$ proposed in [9]. This constraint system is such that:

- Σ is given by the constants symbols $0, 1, 2, \ldots, max-1$,

and the relation symbols $=, \neq, <, \leq, >, \geq$.
- Δ is given by the axioms in number theory.

where $max > 0$. The intended meaning of **FD**$[max]$ is that variables range over a finite domain of values $\{0, \ldots, max - 1\}$. Throughout this paper a **FD** constraint system D is assumed.

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 $\mathcal{C}$.

Process Syntax. Processes communicate with each other by posting and reading partial information (constraints) about the variables of the system they model. This partial information resides in common store of constraints. Henceforth the conjunction of all posted constraints will be simply called *the store*.

Proc is defined as the set of all `rtcc` processes. 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:

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

Intuitively, the process **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. $d \not\vDash c$). 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 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) + $\ldots$, (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

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

where $I = \{i_1, \ldots, i_n\}$. The symbol "+" is used for binary summations (similar to the choice operator from CCS [10]). If there is no ambiguities, the "**when** c **do**" can be omitted when $c = \texttt{true}$, 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 [11]. 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** ($\ldots$ (**next** P) $\ldots$)). A bounded replication and asynchrony can be specified using summation and product. $!_I P$ and $\star_I P$ are defined as abbreviations for $\prod_{i \in I} \textbf{next}^i P$ and $\sum_{i \in I} \textbf{next}^i P$, 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 II.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 later 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:

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

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:

$$\begin{aligned} M_2 \overset{\text{def}}{=} &\ \mathbf{!}\ \mathbf{when}\ action1 \neq 0\ \mathbf{do} \\ &\qquad \mathbf{catch}\ action2 \neq 0 \\ &\qquad\qquad \mathbf{in}\ FollowingActions_{(action1)} \\ &\qquad\qquad \mathbf{finally}\ FollowingActions_{(action2)} \end{aligned}$$

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

III. 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 $\rightarrow\ = \{\xrightarrow{\langle r\rangle}, r \in \mathbb{Z}^+\}$ and $\Rightarrow$ are the least relations satisfying the rules in tables I and II.

The *internal* transition rule $\langle P, d, t\rangle \xrightarrow{r} \langle P', d', t'\rangle$ 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 are not relevant.

The *observable* transition rule $P \xRightarrow{(\iota, o)} Q$ 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.

TABLE I
INTERNAL TRANSITION RULES OF RTCC

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

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

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

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

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

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

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

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

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

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

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

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

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

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

TABLE II
OBSERVABLE TRANSITION RULE OF RTCC

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

Now we are going to explain the transitions rules in tables I and II.

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 [12], 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** $x, \texttt{true}$ **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 [3], 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:

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

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) $P \parallel \textbf{skip} \equiv \textbf{skip} \parallel P \equiv P$
3) $P \parallel Q \equiv Q \parallel P$
4) $\textbf{next skip} \equiv \textbf{skip}$
5) $\textbf{local } x \textbf{ in skip} \equiv \textbf{skip}$
6) $\textbf{local } x\ y \textbf{ in } P \equiv \textbf{local } y\ x \textbf{ in } P$
7) $\textbf{local } x \textbf{ in next } P \equiv \textbf{next}(\textbf{local } x \textbf{ in } P)$

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

Example III.1. To illustrate a sequence of transitions in rtcc, we consider the example II.1. The two machines were modeled thus:

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

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

In order to describe the transitions we have to make some assumptions about the system. Let us say that

- the set *Actions* is composed by 1 and 2, that is, $Actions = \{1, 2\}$,
- the time needed by the constraint system to post a constraint is 2 ticks (no matter the size of the store), and the same for querying the store,
- process *FollowingActions* posts the following four constraints: $reaction1 = 1$, $reaction2 = 2$, $reaction3 = 3$, and $reaction4 = 4$ given the input $action1$ (each posting every two ticks) and posts the constraint $reaction5 = 5$ given the input $action2$, and it consumes only one resource,
- finally, there are 3 resources available (enough for the whole system).

Now, to simplify notation we define

$$P_1 \overset{\text{def}}{=} \sum_{i \in Actions} \mathbf{tell}(action1 = i)$$

$$P_2 \overset{\text{def}}{=} \sum_{i \in Actions} \mathbf{tell}(action2 = i)$$

$$P_3 \overset{\text{def}}{=} \mathbf{delay}\ P_2\ \mathbf{for}\ 5$$

$$P_4 \overset{\text{def}}{=} \mathbf{catch}\ action2 \neq 0\ \mathbf{in}\ FollowingActions_{(action1)}\ \mathbf{finally}\ FollowingActions_{(action2)}$$

$$P_5 \overset{\text{def}}{=} \mathbf{when}\ action1 \neq 0\ \mathbf{do}\ P_4$$

The initial configuration consists in both process M_1 and M_2 executing in parallel, an empty store, and the duration of the time unit (15 ticks). Then internal transitions may look like this for those time units where the first machine performs a single action (since this is nondeterministic it is not possible to know the results in advance):

$$\langle M_1 \parallel M_2, \mathtt{true}, 15\rangle$$

$$\xrightarrow{0} \langle((P_1 \parallel \mathbf{next}\ \mathbf{!}P_1) \parallel \mathbf{next}^m\ P_3) \parallel (P_5 \parallel \mathbf{next}\ \mathbf{!}P_5), \mathtt{true}, 15\rangle$$

$$\xrightarrow{1} \langle(((\mathbf{tell}(action1 = 1) \parallel \mathbf{next}\ \mathbf{!}P_1) \parallel \mathbf{next}^m\ P_3) \parallel (P_5 \parallel \mathbf{next}\ \mathbf{!}P_5), \mathtt{true}, 13\rangle$$

$$\xrightarrow{1} \langle(((\mathbf{skip} \parallel \mathbf{next}\ \mathbf{!}P_1) \parallel \mathbf{next}^m\ P_3) \parallel (\mathbf{when}\ action1\ \mathbf{do}\ P_4 \parallel \mathbf{next}\ \mathbf{!}P_5), action1 = 1, 11\rangle$$

$$\xrightarrow{1} \langle(\mathbf{next}\ \mathbf{!}P_1 \parallel \mathbf{next}^m\ P_3) \parallel (\mathbf{catch}\ action2\ \mathbf{in}\ FollowingActions_{(action1)}\ \mathbf{finally}\ FollowingActions_{(action2)} \parallel \mathbf{next}\ \mathbf{!}P_5), action1 = 1, 9\rangle$$

$$\xrightarrow{1} \langle(\mathbf{next}\ \mathbf{!}P_1 \parallel \mathbf{next}^m\ P_3) \parallel (\mathbf{catch}\ action2\ \mathbf{in}\ FollowingActions_{(action1)}\ \mathbf{finally}\ FollowingActions_{(action2)} \parallel \mathbf{next}\ \mathbf{!}P_5), action1 = 1 \wedge reaction1 = 1, 7\rangle$$

$$\xrightarrow{1} \langle(\mathbf{next}\ \mathbf{!}P_1 \parallel \mathbf{next}^m\ P_3) \parallel (\mathbf{catch}\ action2\ \mathbf{in}\ FollowingActions_{(action1)}\ \mathbf{finally}\ FollowingActions_{(action2)} \parallel \mathbf{next}\ \mathbf{!}P_5), action1 = 1 \wedge reaction1 = 1 \wedge reaction2 = 2, 5\rangle$$

$$\xrightarrow{1} \langle(\mathbf{next}\ \mathbf{!}P_1 \parallel \mathbf{next}^m\ P_3) \parallel (\mathbf{catch}\ action2\ \mathbf{in}\ FollowingActions_{(action1)}\ \mathbf{finally}\ FollowingActions_{(action2)} \parallel \mathbf{next}\ \mathbf{!}P_5), action1 = 1 \wedge reaction1 = 1 \wedge reaction2 = 2 \wedge reaction3 = 3, 3\rangle$$

$$\xrightarrow{1} \langle(\mathbf{next}\ \mathbf{!}P_1 \parallel \mathbf{next}^m\ P_3) \parallel \mathbf{next}\ \mathbf{!}P_5, action1 = 1 \wedge reaction1 = 1 \wedge reaction2 = 2 \wedge reaction3 = 3 \wedge reaction4 = 4, 1\rangle$$

$$\not\rightarrow$$

For those time units where the first machine performs two actions, the internal transitions may look like this:

$$\langle M_1 \parallel M_2, \mathtt{true}, 15\rangle$$

$$\xrightarrow{0} \langle((P_1 \parallel \mathbf{next}\ \mathbf{!}P_1) \parallel P_3) \parallel (P_5 \parallel \mathbf{next}\ \mathbf{!}P_5), \mathtt{true}, 15\rangle$$

$$\xrightarrow{1} \langle(((\mathbf{tell}(action1 = 1) \parallel \mathbf{next}\ \mathbf{!}P_1) \parallel \mathbf{delay}\ P_2\ \mathbf{for}\ 5) \parallel (P_5 \parallel \mathbf{next}\ \mathbf{!}P_5), \mathtt{true}, 13\rangle$$

$$\xrightarrow{1} \langle(((\mathbf{skip} \parallel \mathbf{next}\ \mathbf{!}P_1) \parallel \mathbf{delay}\ P_2\ \mathbf{for}\ 5) \parallel (\mathbf{when}\ action1\ \mathbf{do}\ P_4 \parallel \mathbf{next}\ \mathbf{!}P_5), action1 = 1, 11\rangle$$

$$\begin{aligned}
&\xrightarrow{1} \langle ((\textbf{next } \mathbf{!}P_1 \parallel \textbf{delay } P_2 \textbf{ for } 5) \parallel \\
&\quad (\textbf{catch } action2 \textbf{ in } FollowingActions_{(action1)} \\
&\quad\quad \textbf{finally } FollowingActions_{(action2)} \parallel \\
&\quad \textbf{next } \mathbf{!}P_5), \\
&\quad action1 = 1, 9\rangle \\
&\xrightarrow{1} \langle ((\textbf{next } \mathbf{!}P_1 \parallel \textbf{tell}(action2 = 2)) \parallel \\
&\quad (\textbf{catch } action2 \textbf{ in } FollowingActions_{(action1)} \\
&\quad\quad \textbf{finally } FollowingActions_{(action2)} \parallel \\
&\quad \textbf{next } \mathbf{!}P_5), \\
&\quad action1 = 1 \wedge reaction1 = 1, 7\rangle \\
&\xrightarrow{1} \langle ((\textbf{next } \mathbf{!}P_1 \parallel \textbf{skip}) \parallel \\
&\quad (\textbf{catch } action2 \textbf{ in } FollowingActions_{(action1)} \\
&\quad\quad \textbf{finally } FollowingActions_{(action2)} \parallel \\
&\quad \textbf{next } \mathbf{!}P_5), \\
&\quad action1 = 1 \wedge reaction1 = 1 \wedge reaction2 = 2 \wedge \\
&\quad action2 = 2, 5\rangle \\
&\xrightarrow{1} \langle \textbf{next } \mathbf{!}P_1 \parallel \\
&\quad (FollowingActions_{(action2)} \parallel \textbf{next } \mathbf{!}P_5), \\
&\quad action1 = 1 \wedge reaction1 = 1 \wedge reaction2 = 2 \wedge \\
&\quad action2 = 2, 3\rangle \\
&\xrightarrow{1} \langle \textbf{next } \mathbf{!}P_1 \parallel \textbf{next } \mathbf{!}P_5, \\
&\quad action1 = 1 \wedge reaction1 = 1 \wedge reaction2 = 2 \wedge \\
&\quad action2 = 2 \wedge reaction5 = 5, 1\rangle \\
&\not\rightarrow
\end{aligned}$$

The sequence of observable transitions may look like this:

$$M_1 \parallel M_2 \xRightarrow{\langle \iota_1, o_1\rangle} (\mathbf{!}P_1 \parallel \textbf{next}^2\ P_3) \parallel \mathbf{!}P_5 \xRightarrow{\langle \iota_2, o_2\rangle}$$
$$(\mathbf{!}P_1 \parallel \textbf{next } P_3) \parallel \mathbf{!}P_5 \xRightarrow{\langle \iota_3, o_3\rangle} (\mathbf{!}P_1 \parallel P_3) \parallel \mathbf{!}P_5 \xRightarrow{\langle \iota_4, o_4\rangle} \ldots$$

where each input/output $\langle \iota_i, o_i\rangle$ depends on the choices made; for example if we consider the same internal transitions above we can have

$$\begin{aligned}
\iota_1 &= \langle \texttt{true}, 15, 3\rangle \\
o_1 &= \langle action1 = 1 \wedge reaction1 = 1 \wedge reaction2 = 2 \wedge \\
&\quad reaction3 = 3 \wedge reaction4 = 4, 1, 1\rangle
\end{aligned}$$

$$\begin{aligned}
\iota_4 &= \langle \texttt{true}, 15, 3\rangle \\
o_4 &= \langle action1 = 1 \wedge reaction1 = 1 \wedge reaction2 = 2 \wedge \\
&\quad action2 = 2 \wedge reaction5 = 5, 1, 1\rangle
\end{aligned}$$

□

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 (resources and time) 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, $e \models d$) and restartability explained in [12] 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 III.2. (Internal Extensiveness). *If* $\langle P, c, t\rangle \rightarrow \langle Q, d, t'\rangle$ *then* $d \models c$ *and* $t > t' \geq 0$.

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

The property above can be extended to the observable relation.

Property III.3. (Observable Extensiveness). *If* $P \xRightarrow{(\langle c,r,t\rangle,\ \langle d,s,t'\rangle)} Q$ *then* $d \models c$ *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

$$\langle P_1, c_1, t_1\rangle \rightarrow \langle P_2, c_2, t_2\rangle \rightarrow \ldots \rightarrow \langle P_n, c_n, t_n\rangle \not\rightarrow$$

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 III.2 $c_n \models \ldots \models c_2 \models c_1$ and $t_1 > \ldots > t_n \geq 0$. Hence $d \models c$ 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 $P_1 \stackrel{\text{def}}{=} \textbf{tell}(x = 0)$ and $P_2 \stackrel{\text{def}}{=} \textbf{catch } x = 0 \textbf{ in } Q_1 \textbf{ finally } Q_2$. 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 III.4. *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 III.2. ■

IV. Encoding ntcc into rtcc

The rtcc calculus is an extension of the ntcc calculus. In this sense all systems that can be modeled in ntcc can also be modeled in rtcc (but not in the opposite way). This section is devoted to show this.

In ntcc the notion of time is different from that of rtcc. Each time unit is identified with the time needed for the processes to terminate their computations. In the rtcc calculus we consider that each time unit is identified with the duration given by the environment. Therefore, in ntcc every enabled process computes to its resting point. In our calculus this could be possible in two ways: (1) if we consider a duration of each time unit large enough to have no worries about time, or (2) if the environment knows exactly how much time take each process to get its resting point. The first approach is weak and not rigorous because it is not possible to ensure a good choice for duration. The second choice is more realistic since in the last section we declared two functions to approximate the time spent by a process in two actions. Then if we assume a prior knowledge of processes, we can model in rtcc the systems modeled in ntcc.

On the other hand, the ntcc calculus lacks the notion of resource. This can be solved considering a single resource and relate it with the duration of each time unit.

About the processes themselves, if we do not consider the new constructs (strong time-out and delta delay), most properties of ntcc hold. The reader may observe, for instance, that the strong time-out construct can stop the execution of a process at any time (within a time unit). That is, for a given store a process may evolve, but if that particular store is augmented, it is possible that now the signal that stops the process be present, so the process may evolve in a different way. This leads to think that if there is a catch-free process (i.e. processes without occurrences of the strong time-out construct) which in its execution possibly add some constraints to the store, then it can be executed in the resulting store obtaining the same result.

V. Concluding Remarks and Related Work

In this paper we described the operational semantics of the rtcc calculus. This calculus belongs to the ccp family and is an strict extension of the ntcc calculus. rtcc extends ntcc to allow modeling systems with real-time behaviour. There exists several process calculi that have been extended to support real-time, for instance ACP in [13], CCS in [14] and CSP in [15]. The π-calculus has been extended with real time in two ways: with true concurrency semantics in [16] (stochastic π-calculus), and with interleaving semantics in [17] (the πRT-calculus). Additionally, in the ccp family various extensions have been proposed, for example TScc in [18] and tccp in [19].

In order to guarantee real-time behaviour the operational semantics has a more realistic notion of time than any other ccp-based formalism and includes a transition system with support for expressing amount of resources and time allowance. The notion of resource and its use as a limit for processes have been previously included in various formalisms. Damas P. Gruska in [5], [20] presented an extension of CCS called CCSLP, *Calculus of Communicating Systems with Limited Parallelism*. Patrice Brémond-Grégoire in [21], [8] proposed ACSR, *Algebra of Communicating Shared Resources*. Mikael Buchholtz in [7] presented a process algebra for shared processors.

In computer science talking about real-time always comes with the running time of processes and it depends on the place where they are being executed. Having this in mind, the development of this formal model to support real-time systems such as those of improvisation, lead us to build a theory combining a notion of time as a set of intervals (time units) and a set of points in those intervals (ticks), with a notion of resources as a natural number bounding the concurrency of processes.

We showed the applicability of rtcc by modeling an improvisation system. Then we illustrated the use of the operational semantics by describing a possible sequence of transitions of the system modeled. Previously in [22] we showed the musical expressiveness of the rtcc calculus by modeling musical dissonances.

References

[1] G. Sarria and C. Rueda, "Real-time concurrent constraint programming," in *34th Latinamerican Conference on Informatics (CLEI2008)*, Santa Fe, Argentina, September 2008.

[2] G. Sarria, "Improving the real-time concurrent constraint calculus with a delay declaration," in *Lecture Notes in Engineering and Computer Science: Proceedings of the World Congress on Engineering and Computer Science 2010, WCECS 2010*, San Francisco, California, USA, 20-22 October 2010, pp. 9–14.

[3] V. A. Saraswat, *Concurrent Constraint Programming*, ser. ACM Doctoral Dissertation Award. Cambridge, MA, USA: The MIT Press, 1993.

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

[5] D. P. Gruska, "Process algebra for limited parallelism," in *Concurrency Specification and Programming (CS&P'96)*, Humboldt University, Berlin, 1996, pp. 61–74.

[6] F. S. de Boer, M. Gabbrielli, and M. C. Meo, "A timed concurrent constraint language," *Information and Computation*, vol. 161, no. 1, pp. 45–83, 2000.

[7] M. Buchholtz, J. Andersen, and H. H. Løvengreen, "Towards a process algebra for shared processors," *Electronic Notes in Theoretical Computer Science*, vol. 52, no. 3, 2002.

[8] P. Brémond-Grégoire and I. Lee, "A process algebra of communicating shared resources with dense time and priorities," *Theoretical Computer Science*, vol. 189, no. 1–2, pp. 179–219, December 1997.

[9] P. V. Hentenryck, V. Saraswat, and Y. Deville, "Design, implementation, and evaluation of the constraint language `cc(fd)`," in *Constraint Programming: Basics and Trends, Châtillon Spring School, Châtillon-sur-Seine, France, May 16 - 20, 1994, Selected Papers*, ser. Lecture Notes in Computer Science, A. Podelski, Ed., vol. 910. Springer-Verlag, 1995, pp. 293–316.

[10] R. Milner, *A Calculus of Communicating Systems*, ser. Lecture Notes in Computer Science. Springer-Verlag, 1980.

[11] C. A. R. Hoare, *Communicating Sequential Processes*, ser. Prentice-Hall International Series in Computer Science. Prentice Hall, April 1985.

[12] F. S. de Boer, A. D. Pierro, and C. Palamidessi, "Nondeterminism and infinite computations in constraint programming," in *Selected Papers of the Workshop on Topology and Completion in Semantics*, ser. Theoretical Computer Science, vol. 151, no. 1. Chartres, France: Elsevier Science Publishers B. V., 1995, pp. 37–78.

[13] J. Baeten and J. A. Bergstra, "Real time process algebra," *Formal Aspects of Computing*, vol. 3, no. 2, pp. 142–188, 1991.

[14] C. J. Fidge, "A constraint-oriented real-time process calculus," in *IFIP TC6/WG6.1 Fifth International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE'92)*, M. Diaz and R.Groz, Eds., Lannion, France, October 1992, pp. 355–370.

[15] J. Davies, "Specification and proof in real-time csp," Ph.D. dissertation, University of Oxford, 1993.

[16] C. Priami, "Stochastic π-calculus," *The Computer Journal*, vol. 38, no. 7, pp. 578–589, 1995.

[17] J. Y. Lee and J. Zic, "On modeling real-time mobile processes," in *Twenty-fifth Australasian Conference on Computer Science (ACSC'02)*, ser. Conferences in Research and Practice in Information Technology, vol. 17. Melbourne, Victoria, Australia: Australian Computer Society, Inc., 2002, pp. 139–147.

[18] L. Brim, M. Křetinský, D. Gilbert, and J.-M. Jacquet, "Temporal synchronous concurrent constraint programming," in *1st International Workshop on Constraint Programming for Time Critical Systems (COTIC'97)*, Piza, Italy, October 1997, pp. 35–50.

[19] F. S. de Boer, M. Gabbrielli, and M. C. Meo, "A temporal logic for reasoning about timed concurrent constraint programs," in *TIME*, 2001, pp. 227–233.

[20] D. P. Gruska, "Bounded concurrency," in *11th International Symposium on Fundamentals of Computation Theory (FCT'97)*, ser. Lecture Notes In Computer Science, vol. 1279. London, UK: Springer-Verlag, 1997, pp. 198–209.

[21] P. Brémond-Grégoire, H. Ben-Abdallah, and I. Lee, "Ordering processes in a real-time process algebra," in *3rd International Workshop on Real-Time Systems (AMAST'96)*, Red Lion Hotel, Salt Lake City, Utah, USA, March 1996.

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