Real-Time Concurrent Constraint Programming

Gerardo M. Sarria M. and Camilo Rueda

Pontificia Universidad Javeriana
Cali - Colombia

{gsarria,crueda}@cic.puj.edu.co

Abstract. The ntcc calculus is a model of temporal concurrent constraint programming with the capability of expressing asynchronous and non-deterministic timed behaviour. We propose a model of real-time concurrent constraint programming, which adds to ntcc the means for specifying and modelling real-time behaviour. We provide a new construct for strong preemption, an operational semantics supporting resources and limited time and a denotational semantics based on CHU spaces. We argue that the resultant calculus has a natural application in various fields such as multimedia interaction.

1 Introduction

The ntcc calculus [1] is a model of temporal concurrent constraint programming with the capability of modeling timed behaviour. It extends the tcc calculus [2], a model of reactive systems, with the notions of asynchrony and nondeterminism.

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 ntcc the stimulus is a constraint representing the initial store and the response is another constraint representing the final store after calculations. A reactive system is shown in figure 1. For each P_i there is an stimulus d_i and a response d'_i in the time unit t_i.

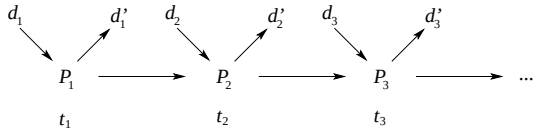

Fig. 1. Reactive System

Time in ntcc is viewed as a sequence of time slots where each time unit is identified with the time needed for a process to terminate a computation. In other words, each time unit is defined by the time taken by all processes to make all their internal transitions until no further transition can be done. This is not enough to satisfy quantitative temporal constraints which is a requirement of real-time

systems (e.g. music improvisation). Therefore, in this paper we propose `rtcc`, an extension of `ntcc` with constructs for expressing real-time behaviour.

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 given by 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 corresponds exactly to two moments in the physical time).

Now, most formal models of processes abstract away many real properties of existing systems such as duration of actions and number of processors [3]. Others assume maximal parallelism, that is the assumption of having n processors to execute n parallel processes (as in [4]). Nevertheless, for real-time systems the fact that processes have to share one processor cannot be ignored, since it may influence both the timely and the functional behaviour of the system [5]. Moreover, as it is said in [6] the timing 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, the transition system is extended to describe sharing of resources. We assume that the environment provides a number r of resources. Each process P takes some of these. When P is finished, it releases them.

On the other hand, an essential issue in reactive and real-time systems is process preemption. In [7], this concept is defined as the control mechanism consisting in denying a process the right to work, either permanently (abortion) or temporarily (suspension).

In music improvisation situations, for instance, there are cases in which the musician must skip some note or play something different to synchronize with the other members of the band, or wait for a signal or for some time before continuing with his part. These examples show temporal requirements and the need of satisfying some temporal constraints that involve a set of processes.

There are two ways to preempt a process: bounding its execution time or with some signal. For the first one, we assume that the environment also 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 second way of preempt a process is aborded with the constructs of the calculus.

A reactive system is now described graphically in figure 2. A process receives as stimulus the initial store, the available number of resources and the duration of the time unit, and responds with the final store, the maximum number of resources used in calculations and the time spent in them.

The main contributions of this paper are: 1) the definition of `rtcc`, a real-time concurrent constraint programming extension of the `ntcc` calculus, with a construct to express strong preemption, 2) an extension of the transition system supporting resources and limited time, and 3) a denotational semantics based on CHU spaces.

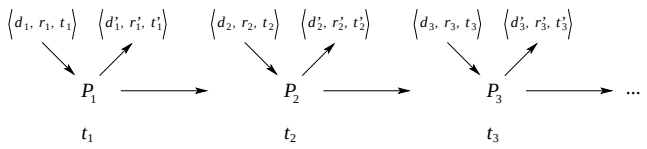

Fig. 2. Reactive System in `rtcc`

2 The `rtcc` Calculus

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:

$$P, Q, \ldots ::= \mathbf{tell}(c) \mid \textstyle\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i \mid P \parallel Q \mid \mathbf{local}\ x\ \mathbf{in}\ P \mid \mathbf{next}\ P \\ \mid \mathbf{unless}\ c\ \mathbf{next}\ P \mid \mathbf{catch}\ c\ \mathbf{in}\ P\ \mathbf{finally}\ Q \mid\ !P \mid \star P$$

Intuitively, the process $\mathbf{tell}(c)$ adds constraint c to the store within the current time unit. The ask process $\mathbf{when}\ c\ \mathbf{do}\ P$ is generalized with a non-deterministic choice of the form $\sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i$ (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 $\mathbf{local}\ x\ \mathbf{in}\ P$ 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 execution of a process P can be delayed with $\mathbf{next}\ P$. The process P will be activated in the next time interval. The weak time-out process, $\mathbf{unless}\ c\ \mathbf{next}$

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. $d \not\models c$). 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 (this is similar to the construct "**do** A **watching immediately** c" of ESTEREL [8]). When process P is interrupted, process Q is executed. If P finishes, Q is discarded. The operator "**!**" is used to define infinite behaviour. The process $!P$ represents $P \parallel \mathbf{next}\ P \parallel \mathbf{next}(\mathbf{next}\ P) \parallel \ldots$, (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 $P + \mathbf{next}\ P + \mathbf{next}(\mathbf{next}\ P) + \ldots$, (i.e. it allows to model asynchronous behaviour across the time intervals).

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

Example 1. Suppose a simple improvisation situation where there are two musicians M_1 and M_2. The first musician M_1 plays a single random note from a list $Notes$. The second musician M_2 must adorn it, that is, play a series of chords depending on the note played by M_1. Additionally, in some occasions M_1 not only plays a single note but two in the same time unit. In this case M_2 must stop his performance and try to adorn the second note (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:

$$M_1 \stackrel{\text{def}}{=} !\sum_{i \in Notes} \mathbf{tell}(note1 = i) \parallel \star \sum_{i \in Notes} \mathbf{tell}(note2 = i)$$

Now for the second musician a process $Adorn$ that calculate the musical adornment and perform it is assumed. Thus M_2 is modeled:

$$M_2 \stackrel{\text{def}}{=} !\ \mathbf{when}\ note1\ \mathbf{do}\ (\mathbf{catch}\ note2\ \mathbf{in}\ Adorn_{(note1)}\ \mathbf{finally}\ Adorn_{(note2)})$$

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 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 table 1.

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 are not relevant. The *observable* transition rule $P \overset{(\iota,o)}{\Longrightarrow} Q$ 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. Intuitively, a time unit has a certain duration (this duration depends on the time taken by the constraint system to answer ask and tell operations). An observable transition is constructed from a sequence of internal transitions. It is assumed that internal transitions cannot be directly observed.

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

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

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

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

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

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

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

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

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

$$\frac{\langle P, c, t\rangle \xrightarrow{s}{}^{*} \langle Q, d, t'\rangle \not\to}{P \xrightarrow{(\langle c,r,t\rangle,\ \langle d,s,t-t'\rangle)} R} \quad \text{if } R \equiv F(Q)$$

Table 1. Transition Rules of `rtcc`

Now we are going to explain the transitions rules in table 1. 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 a function $\Phi : |D|_0 \times |D|_0 \longrightarrow \mathbb{N} - \{0\}$ ($\Phi(c,d)$ approximates the time spent in adding constraint c to store d, for simplicity, the same function is used for estimating the time querying the store). 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, except that it has to take into account all the possible branches of the process (since this is a non-deterministic choice, nobody knows what process will be chosen, so the store must be queried with every guard). 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.

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 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

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

To simplify the transitions, the congruence relation $\equiv$ is defined as follows: 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. $\mathbf{catch}\ c\ \mathbf{in}\ (P \parallel Q) \equiv \mathbf{catch}\ c\ \mathbf{in}\ P \parallel \mathbf{catch}\ c\ \mathbf{in}\ Q$

4 Denotational Semantics

The denotational semantics of the processes in the `rtcc` calculus is defined by giving the *denotations* of the processes. A denotation is a labelled K-valued Chu space (with $|K| = 4$) where the events can be thought of as the acting of adding information to the store and its labels are the actual information it adds.

A Chu space [9] is simply a $|A|\times|X|$ rectangular matrix over a given set K (the alphabet). Formally, a Chu space can be viewed as a tuple $\mathcal{C} = \langle A, X, R\rangle$, over a set K, where A (the rows) is a set of events, X (the columns) is a set of states, and $R : A \times X \times K$ is a relation constituting the matrix. To consider the actions that a process executes, a labeled Chu space is defined as the triple (A, X, λ), where $\lambda : A \to Act$ is the labeling function (Act is a set of possible actions). The reader may consult [9] for further details on CHU spaces and its algebra.

We must remark a distinction between *events* and *actions*. The occurrence of an event performs an action. In this way, two different events might perform the same action. In this calculus an event may be thought as the addition of information. Its action is the actual information it adds.

As defined in [10] the elements of K are the possible values of an event in a given state, with 0, $\lrcorner\!\ulcorner$, 1, $\times$ corresponding to *before*, *during*, *after* and *instead*. In a given state an event has value 0 when it has not yet started, $\lrcorner\!\ulcorner$ when it is happening, 1 when it has finished, and $\times$ when it has been canceled.

To define the denotations, let $\mathcal{P} = (A, X, \lambda_1)$ and $\mathcal{Q} = (B, Y, \lambda_2)$ be Chu spaces over K, giving semantics to `rtcc` processes P and Q, respectively. We use the semantic braces $[\![\ ]\!]$ to denote a function that associates a process to a Chu space. The denotation of $\mathbf{tell}(c)$ is given by

$$[\![\mathbf{tell}(c)]\!] = c\boxed{0\ \lrcorner\!\ulcorner\ 1} \qquad \lambda(c) \mapsto c$$

This states that the semantic definition of a tell operation is a Chu space with 3 states: when the process has not yet added the information, when it is adding the constraint, and when it has already posted the constraint. The denotation of $\mathbf{tell}(c)$ represents, in the state x where $x(c) = 1$ (the third state in the Chu space), all the possible stores which contain at least as much information as c, given an arbitrary input from the environment.

The operation $\mathbf{when}\ c\ \mathbf{do}\ P$, can be seen as a sequential process. Information c should be added before P can execute. Thus, the denotation for ask operators is defined by

$$[\![\mathbf{when}\ c\ \mathbf{do}\ P]\!] = [\![\mathbf{tell}(c)]\!]\ ;\ \mathcal{P}$$

For any two processes $\mathcal{P}$, $\mathcal{Q}$ the sequence $\mathcal{P}\ ;\ \mathcal{Q}$ is defined as $\mathcal{P} \wedge \mathcal{Q} \wedge (\overline{B} \vee \checkmark\mathcal{P})$. $\mathcal{P} \wedge \mathcal{Q}$ is defined as the union of the set of events of both CHU spaces restricting those states in P and Q. Notation $\overline{B}$ (also written $B = 0$) denotes the process $(B, \{0^B\})$ having just the one all-zero state. Additionally, the Chu space $\checkmark\mathcal{P}$

denotes a process whose events have already occurred. The non-deterministic choice (summation) of two processes has the following denotation

$$[\![P + Q]\!] = \overline{A \cup B} \vee (\mathcal{P} \neq 0 \wedge \overline{B - A}) \vee (\mathcal{Q} \neq 0 \wedge \overline{A - B})$$

where $\mathcal{P} \neq 0$ denotes $(A, X - \{0^A\})$. So, events of $\mathcal{P}$ and $\mathcal{Q}$ might have not started yet ($\overline{A \cup B}$), or P have started but those events in B not in A have not occurred ($\mathcal{P} \neq 0 \wedge \overline{B - A}$), or similarly for Q.

Parallel composition $P \parallel Q$ is the joint behaviour of P and Q. Then

$$[\![P \parallel Q]\!] = \mathcal{P} \wedge \mathcal{Q}$$

The semantics for a local behaviour is

$$[\![\textbf{local } x \textbf{ in } P]\!] = (A, X, \lambda_k) \quad \text{where } \forall a \in A \;.\; \lambda_k(a) = \exists x.\lambda_1(a)$$

For each event in process $\mathcal{P}$, the variable x in the labeling function λ_1 is local to it. The semantics for a strong time-out is

$$[\![\textbf{catch } c \textbf{ in } P \textbf{ finally } Q]\!] = (\surd[\![\textbf{tell}(c)]\!] \wedge (\overline{A} \vee \not{A}) \wedge \mathcal{P}) \vee ([\![\textbf{tell}(c)]\!] = 0 \wedge \mathcal{P})$$

Here $\not{A}$ denotes the process $\mathcal{P} = \times$ (which is $(A, \{\times^A\})$), that is, all events in process $\mathcal{P}$ are canceled. This process has two possible behaviours: if the constraint c has been already posted, then either process P has not started or it has been canceled ($\surd[\![\textbf{tell}(c)]\!] \wedge (\overline{A} \vee \not{A})$), or the constraint c has not been added to the store yet and the process P continues its execution ($[\![\textbf{tell}(c)]\!] = 0 \wedge \mathcal{P}$).

The following are the denotations for the temporal constructs. For these denotations we assume a process $CLOCK$ of the form $\boxed{0 \lrcorner 1}$ for a unique event *clock*, distinct from all other events. Process $CLOCK$ is always active. This means that once a time unit is over, the event of $CLOCK$ changes its value to 1, but just before the next time unit starts, it starts again with 0 (the sequence $CLOCK \;;\; CLOCK$ distinguish the events, which perform the same action). Hence, all processes are executed in parallel with the event of $CLOCK$ (the current time unit).

The unit delay construct **next** P will execute process P the next time unit. Hence, its denotation is the CHU space representing the sequence of processes $CLOCK$ and $\mathcal{P}$. This gives the following semantic equation:

$$[\![\textbf{next } P]\!] = CLOCK \;;\; \mathcal{P}$$

The process **unless** c **next** P have the following denotation

$$[\![\textbf{unless } c \textbf{ next } P]\!] = ([\![\textbf{tell}(c)]\!] = 0 \wedge CLOCK \;;\; \mathcal{P}) \vee ([\![\textbf{tell}(c)]\!] \neq 0)$$

Either process $\mathcal{P}$ will be executed after process $CLOCK$ (similar to the denotation of next) but if c is not added in the current time unit, or c is being added or has been added then P will not be executed.

The denotation for replication $!P$ can be defined recursively as the denotation of P related with the process $CLOCK$, followed by the same denotation of $!P$.

$$[\![!P]\!] = \mathcal{P} \sqcup CLOCK \;;\; [\![!P]\!]$$

where process $\mathcal{P} \sqcup CLOCK$ is defined as $\overline{CLOCK \vee \mathcal{P}} \ \vee \ (CLOCK = \lrcorner \wedge \mathcal{P}) \vee (\checkmark CLOCK \wedge \mathcal{P} \neq \lrcorner)$.

Recursive definitions of CHU spaces such as the one above have been proven to exists. The reader may consult the proof in [9].

Asynchrony is defined as the bounded delay of a process. If m is an arbitrary nonnegative integer then the denotation of $\star P$ is the denotation of $\mathbf{next}^m\ P$:

$$[\![\star P]\!] = [\![\mathbf{next}^m\ P]\!] \quad \text{where } m > 0$$

$$\text{where } [\![\mathbf{next}^m\ P]\!] = \underbrace{CLOCK\ ;\ CLOCK\ ;\ \ldots\ ;\ CLOCK}_{m}\ ;\ P$$

The process will be executed after an arbitrary number of processes $CLOCK$ are executed. Now we give a simple example illustrating the denotations of processes.

Example 2. Let $P \stackrel{\text{def}}{=} \mathbf{catch}\ r\ \mathbf{in}\ \mathbf{tell}(b)$ and $Q \stackrel{\text{def}}{=} \mathbf{tell}(b)$. Then $[\![P \parallel Q]\!]$ is computed thus:

$$[\![P]\!] = [\![\mathbf{catch}\ r\ \mathbf{in}\ \mathbf{tell}(b)]\!] = \left(r\boxed{1} \wedge \left(b_1\boxed{0} \vee b_1\boxed{\times}\right)\right) \vee \left(r\boxed{0} \wedge b_1\boxed{0 \lrcorner 1}\right)$$

$$= \begin{pmatrix} r & \boxed{1\ 1} \\ b_1 & \boxed{0\ \times} \end{pmatrix} \vee \begin{pmatrix} r & \boxed{0\ 0\ 0} \\ b_1 & \boxed{0\ \lrcorner\ 1} \end{pmatrix} = \begin{matrix} r \\ b_1 \end{matrix}\boxed{\begin{matrix} 1 & 1 & 0 & 0 & 0 \\ 0 & \times & 0 & \lrcorner & 1 \end{matrix}} \qquad \begin{matrix} \lambda(r) \mapsto r \\ \lambda(b_1) \mapsto b \end{matrix}$$

$$[\![Q]\!] = [\![\mathbf{tell}(b)]\!] = b_2\boxed{0 \lrcorner 1} \qquad \lambda(b_2) \mapsto b$$

$$[\![P \parallel Q]\!] = \begin{matrix} r \\ b_1 \end{matrix}\boxed{\begin{matrix} 1 & 1 & 0 & 0 & 0 \\ 0 & \times & 0 & \lrcorner & 1 \end{matrix}} \wedge b_2\boxed{0 \lrcorner 1} = \begin{matrix} r \\ b_1 \\ b_2 \end{matrix}\boxed{\begin{matrix} 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & \times & 0 & \lrcorner & 1 & 0 & \times & 0 & \lrcorner & 1 & 0 & \times & 0 & \lrcorner & 1 \\ 0 & 0 & 0 & 0 & 0 & \lrcorner & \lrcorner & \lrcorner & \lrcorner & \lrcorner & 1 & 1 & 1 & 1 & 1 \end{matrix}} \qquad \begin{matrix} \lambda(r) \mapsto r \\ \lambda(b_1) \mapsto b \\ \lambda(b_2) \mapsto b \end{matrix}$$

Note that in the denotation of the process $P \parallel Q$, all the branches of its execution are included (e.g. if both processes are being executed and r becomes present, then the states $\begin{matrix} r \\ b_1 \\ b_2 \end{matrix}\boxed{\begin{matrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & \lrcorner & 0 & \lrcorner & \lrcorner & \times & \times \\ 0 & 0 & \lrcorner & \lrcorner & 1 & \lrcorner & 1 \end{matrix}}$ will be active). Note also that although the events of posting the constraint b by both processes is different (they have two different rows in the CHU space), they perform the same action. Therefore, the labelling function λ of the resulting CHU space applied with the denotation of both events returns the same, i.e. $\lambda(b_1) \mapsto b$, and $\lambda(b_2) \mapsto b$.

5 Concluding Remarks and Future Work

In this paper we proposed a new real-time concurrent constraint calculus called `rtcc`. This calculus allows to model real-time behaviour and specify strong preemption. `rtcc` is obtained from `ntcc` by adding a new constructs and extending the transition system with support for resources and limited time.

We have shown the use of this formal model with a simple improvisation example. Some other non-trivial examples have shown us the necessity of a way to define delays within a time unit. We plan to include a construct for delaying a process within the current time unit. This construct may enhance the calculus by making it temporally homogeneous but it also may lead to some operational problems (such as the activation of a process when the time unit is over) and to a non-correspondence between operational and denotational semantics.

The lack of monotonicity derived from the construct **catch** c **in** P **finally** Q and the inclusion of resources in the operational semantics precludes defining the denotations of processes in terms of quiescent points as is usual in CCP calculi such as `ntcc`. Instead, the denotations were built as CHU spaces. This has shown to be convenient since the algebra associated with this model of concurrency contains the necessary operations to manipulate the processes and to obtain a correspondence with the operational semantics.

For a transition system disregarding resources a real-time logic was defined in [11] based on RTTL [12]. This was not included in this paper due to the lack of space. The presence of resources in the calculus makes difficult to express properties in the logic. The BI logic [13] and the separation logic [14] are the logics handling resources that we know of. We plan to study the application of these logics in our research.

We are interested in using the calculus presented here to model complex improvisation systems and some other music systems such as OpenMusic Maquettes [15]. Then we plan to build a processes simulator for `rtcc` to better visualize the behaviour of these systems and to make possible listening the audio results of the models in real-time.

References

1. C. Palamidessi and F. Valencia. A Temporal Concurrent Constraint Programming Calculus. In *CP'2001*, volume 2239 of *LNCS*, p.302–316, 2001.
2. V.A. Saraswat, R. Jagadeesan and V. Gupta. Foundations of Timed Concurrent Constraint Programming. In *IEEE LICS'94*, p.71–80, 1994.
3. D. P. Gruska. Process Algebra for Limited Parallelism. In *CS&P'96*, pages 61–74, Humboldt University, Berlin, 1996.
4. F. S. de Boer, M. Gabbrielli and M. C. Meo. A Timed Concurrent Constraint Language. *Information and Computation*, 161(1):45–83, 2000.
5. M. Buchholtz, J. Andersen and H.H. Løvengreen. Towards a Process Algebra for Shared Processors. *Electronic Notes in Theoretical Computer Science*, 52(3), 2002.
6. P. Brémond-Grégoire and I. Lee. A Process Algebra of Communicating Shared Resources with Dense Time and Priorities. *Theoretical Computer Science*, 189(1–2):179–219, December 1997.
7. G. Berry. Preemption in Concurrent Systems. In *FSTTCS'93*, p.72–93, 1993.
8. G. Berry and G. Gonthier. The Esterel Synchronous Programming Language: Design, Semantics, Implementation. *Science Compt. Programming*, 19(2):87–152, 1992.
9. V. Gupta. *Chu Spaces: A Model Of Concurrency*. PhD Thesis, Stanford, 1994.
10. V.R. Pratt. Transition and cancellation in concurrency and branching time. *Mathematical Structures in Computer Science*, 13(4):485–529, 2003.
11. G. Sarria. *Formal Models of Timed Musical Processes*. PhD Thesis, Universidad del Valle Cali-Colombia, To appear, 2008.
12. J.S. Ostroff. Temporal Logic for Real-Time Systems. *John Wiley & Sons Inc*, 1989.
13. D. Pym and C. Tofts. A Calculus and Logic of Resources and Processes. *Formal Aspects of Computing*, 18(4):495–517, November 2006.
14. P.W. O'Hearn. Resources, Concurrency, and Local Reasoning. *Theoretical Computer Science*, 375(1–3):271–307, May, 2007.
15. G. Assayag, C. Rueda, M. Laurson, C. Agon and O. Delerue. Computer-Assisted Composition at IRCAM: from PatchWork to OpenMusic. *Computer Music Journal*, 23(3):59–72, Fall 1999.