NTCCRT: A CONCURRENT CONSTRAINT FRAMEWORK FOR REAL-TIME INTERACTION (EXTENDED VERSION)

Mauricio Toro-Bermúdez *Camilo Rueda* *Carlos Agón* *Gérard Assayag*
Pontificia Universidad Javeriana Cali IRCAM IRCAM - CNRS UMR 9912

ABSTRACT

Writing multimedia interaction systems is not easy. Their concurrent processes usually access shared resources in a non-deterministic order, often leading to unpredictable behavior. Using Pure Data (Pd) and Max/MSP is possible to program concurrency, however, it is difficult to synchronize processes based on multiple criteria. Process calculi such as the Non-deterministic Timed Concurrent Constraint (ntcc) calculus, overcome that problem by representing multiple criteria as constraints. We propose using our framework Ntccrt to manage concurrency in Pd and Max. Ntccrt is a real-time capable interpreter for ntcc. Using Ntccrt externals (binary plugins) in Pd we ran models for machine improvisation and signal processing.

1. INTRODUCTION

Multimedia interaction systems –inherently concurrent– can be modeled using concurrent process calculi. Process calculi are useful to describe formally the behavior of concurrent systems, and to prove properties about the systems.

For instance, using `ntcc` [9], we can model reactive systems with synchronous, asynchronous and/or non-deterministic behavior. `Ntcc` and extensions have been used to model interactive systems such as: an audio processing framework [18], machine improvisation [17], [12], [20], and interactive scores [2], [20].

Although there are three interpreters for `ntcc`, they are not suitable for real-time (RT) interaction. It means that they are not able to interact with the user without letting him experience noticeable delays in the interaction.

On the other hand, we can program RT systems for multimedia interaction and signal processing using C++. Unfortunately, using C++ requires long development time. To overcome that problem, programming languages such as Pure Data (Pd) [13] and Max/MSP [14], provide a graphical interface to program RT systems and several APIs for concurrent programming.

1.1. The problem

Although Pd and Max support concurrency, it is a hard task to trigger or halt the execution of a process based on multiple criteria.

Using Pd or Max, it is hard to express: "process *A* is going to do an action *B* until a condition *C* is satisfied", when condition *C* is a complex condition resulting from many other processes' actions. Such condition would be hard to express (and even harder to modify afterwards) using the graphical patch paradigm. For instance, condition *C* can be a conjunction of these criteria: (1) The user has played on a certain tonality, (2) has played the chord G7, and (3) played the note F# among the last four.

1.2. Our solution

Using `ntcc`, we can represent the complex condition *C* presented above as the conjunction of constraints ($c_1 \wedge c_2 \wedge c_3$). Each constraint (i.e., mathematical condition) represents a criterion. In addition, each criterion can be represented declaratively. For instance, the criterion (2) can be represented by the constraint "G7 is on the set of played chords" ($G_7 \in PlayedChords$).

For that reason, we propose using `ntcc` to manage concurrency in Pd and Max, executing `ntcc` models on *Ntccrt*[1]. On *Ntccrt*, `ntcc` models can be automatically compiled as an *external* (i.e., a binary plugin) for Pd or Max.

Additionally, the externals can be specified textually using Common Lisp or graphically using OpenMusic [4]. We argue that concurrent visual programming, usually based on process calculi (such as Cordial [15]), makes the power of concurrency available for a wider range of users.

1.3. Contributions

Our framework *Ntccrt* (http://ntccrt.sourceforge.net) is composed by the following components. The `ntcc` interpreter written in C++ and interfaces for both Common Lisp and OpenMusic. In addition, we provide the implementation of two real-life applications.

1.4. Structure of the paper

The remainder of this paper is structured as follows. Section 2 intuitively explains the semantic of `ntcc` agents and gives

[1] This research was partially founded by the REACT project, sponsored by Colciencias.

some examples of simple ntcc processes modeling multimedia interaction. Section 3 explains related work on ntcc interpreters and threading APIs available for Pd and Max. Section 4 discusses two applications of *Ntccrt* to model a multimedia interaction and a signal processing system. Section 5 explains our results. Finally, section 6 gives concluding remarks and proposes future works.

2. THE NTCC CALCULUS

A family of process calculi is Concurrent Constraint Programming (CCP) [19], where a system is modeled in terms of variables and constraints over some variables. Furthermore, there are agents reasoning about partial information (by the means of constraints) about the system variables contained on a common *store*.

CCP is based on the idea of a *constraint system*. A constraint system includes a set of (basic) constraints and a relation (i.e., entailment relation $\models$) to deduce a constraint based on the information supplied by other constraints. A CCP system usually includes several constraint systems for different variable types. There are constraint systems for different variable types such as sets, trees, graphs and natural numbers. A constraint system providing arithmetic relations over natural numbers is known as Finite Domain (FD). For instance, using a FD constraint system we can deduce the constraint $pitch \neq 60$ from the constraints $pitch > 40$ and $pitch < 59$.

Although we can choose an appropriate constraint system to model any problem, in CCP it is not possible to delete nor change information accumulated in the store. For that reason, it is difficult to perceive a notion of discrete time, useful to model reactive systems (e.g., machine improvisation) communicating with an environment.

Ntcc introduces to CCP the notion of discrete time as a sequence of *time-units*. Each time-unit starts with a *store* (possibly empty) supplied by the environment, then ntcc executes all processes scheduled for that time-unit. In contrast to CCP, in ntcc, variables changing values along time can be modeled explicitly. In ntcc, we can have a variable x taking different values on each time-unit. To model that in CCP, we would have to create a new variable x_i each time we change the value of x.

For instance, a system that plays sequentially the notes of the C major chord can be modeled in ntcc as "in the first time-unit, let $pitch = C$; in the second time-unit, let $pitch = E$; and in the third time-unit, let $pitch = G$". Using CCP, we would represent it as "let $pitch_1 = C$, let $pitch_2 = E$, and let $pitch_3 = G$".

Following, we give some examples of how the computational agents of ntcc can be used with a FD constraint system. A summary can be found in table 1.

Using the "tell", it is possible to add constraints such as **tell**$(pitch_1 = 60)$, meaning that $pitch_1$ must be equal to 60

Agent	Meaning
tell (c)	Adds c to the current store
when (c) **do** A	If c holds now run A
local (x) **in** P	Runs P with local variable x
$A \parallel B$	Parallel composition
next A	Runs A at the next time-unit
unless (c) **next** A	Unless c can be inferred now, run A
$\sum_{i \in I}$ **when** (c_i) **do** P_i	Chooses P_i s.t. (c_i) holds
$*P$	Delays P indefinitely (not forever)
$!P$	Executes P each time-unit

Table 1. Ntcc agents

or **tell**$(60 < pitch_2 < 100)$, meaning that $pitch_2$ is an integer between 60 and 100.

The "when" can be used to describe how the system reacts to different events. For instance, **when** $pitch_1 = C \wedge pitch_2 = E \wedge pitch_3 = G$ **do tell**$(CMayor = true)$ is a process reacting as soon as the pitch sequence C, E, G has been played, adding the constraint $CMayor = true$ to the store in the current time-unit.

Parallel composition allows us to represent concurrent processes. For instance, **tell** $(pitch_1 = 62) \parallel$ **when** $60 \leq pitch_1 < 72$ **do tell** $(Instrument = 1)$ is a process telling the store that $pitch_1$ is 62 and concurrently assigning the instrument to one, since $pitch_1$ is in first octave.

The "next" is useful when we want to model variables changing through time. For instance, **when** $(pitch_1 = 60)$ **do next tell** $(pitch_1 <> 60)$, means that if $pitch_1$ is equal to 60 in the current time-unit, it will be different from 60 in the next time-unit.

The "unless" is useful to model systems reacting when a condition is not satisfied or it cannot be deduced from the store. For instance, **unless** $(pitch_1 = 60)$ **next tell** $(lastpitch <> 60)$ reacts when $pitch_1 = 60$ is false or when $pitch_1 = 60$ cannot be deduced from the store (e.g., $pitch_1$ was not played in the current time-unit), telling the store in the next time-unit that $lastpitch$ is not 60.

The "star" (*) may be used to delay the end of a process indefinitely, but not forever. For instance, $*$**tell** $(End = true)$.

The "bang" (!) executes a certain process in every *time-unit* after its execution. For instance, !**tell** $(C_4 = 60)$.

The $\sum$ is used to model non-deterministic choices. For instance, $! \sum_{i \in \{48,52,55\}}$ **when** $i \in PlayedPitches$ **do tell** $(pitch = i)$ models a system where each time-unit, it chooses a note among the notes played previously that belongs to the C major chord.

Finally, a basic recursion can be defined in ntcc with the form $q(x) \stackrel{def}{=} P_q$, where q is the process name and P_q is restricted to call q at most once and such call must be within the scope of a "next". The reason of using "next" is that

ntcc does not allow recursion within a time-unit. Recursion is used to model iteration and recursive definitions. For instance, using this basic recursion, it is possible to write a function to compute the factorial function.

3. RELATED WORK

In this section, we present related work about concurrency support for Pd and Max, and available ntcc interpreters.

3.1. Writing concurrent programs on Pd and Max

To program concurrent applications on Max and Pd, we can use their message passing APIs. We can also create externals in C++. In fact, we can use any existing threading API for C++ to write externals for both, Pd and Max. There is also a native API for Max 5 SDK. Another way to write an external is using the Flext library. Flext provides a unique interface to write, in the C++ language, externals dealing with both, Pd and Max.

3.2. Ntcc interpreters

There are three interpreters available for ntcc: Lman [8] used as a framework to program Lego^{TM} robots, NtccSim [3] used to model and verify properties of biological systems, and Rueda's interpreter [17] for multimedia interaction.

The first attempt to execute a multimedia interaction ntcc model was made by the authors of Lman in 2003. They ran a ntcc model to play a sequence of pitches with fixed durations in Lman. Recently, in 2006, Rueda et al. ran "A Concurrent Constraint Factor Oracle Model for Music Improvisation" (*Ccfomi*) on Rueda's interpreter [17].

Both, *Lman* and *Rueda's interpreter* ran the model giving the expected output. However, they were not capable of executing multimedia interaction systems in real-time.

4. OUR FRAMEWORK: NTCCRT

Ntccrt is our framework to specify and execute ntcc models.

4.1. Design of Ntccrt

Our first version of *Ntccrt* allowed us to specify ntcc models in C++ and execute them as stand-alone programs. Current version offers the possibility to specify a ntcc model on either Lisp, Openmusic or C++. It is also possible to execute ntcc models as a stand-alone program or as an external object for Pd or Max.

In addition to its portability, *Ntccrt* was carefully designed to support Finite Domain, Finite Sets and Rational Trees constraint systems. Those constraint systems can be used to represents complex data structures (e.g., automata and graphs) commonly used in computer music.

Ntccrt works on two modes, one for writing the models and another one for executing those models.

4.1.1. Developing mode

In order to write a ntcc model in *Ntccrt*, the users may write them directly in C++, using a parser that takes Common Lisp macros or writing a graphical "patch" in Open-Music. Using either of these representations, it is possible to generate a stand-alone program or an external (fig 1).

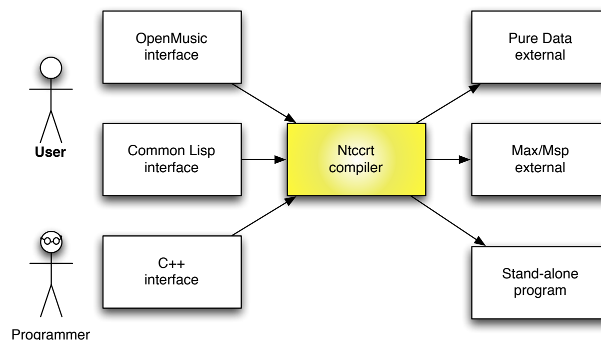

Figure 1. *Ntccrt*: Developing mode

4.1.2. Execution mode

To execute a *Ntccrt* program, we can proceed in two different ways. We can create a stand-alone program or we can create an external for either Pd or Max. An advantage of using the externals lies on using control signals and the message passing API provided by Pd and Max to synchronize any *graphical object* with the *Ntccrt* external.

To handle Musical Instrument Digital Interface (MIDI) streams we use the predefined functions in Pd or Max to process MIDI. Then, we connect the output of those functions to the *Ntccrt* external. We also provide an interface for Midishare [5], useful when running stand-alone programs.

4.2. Implementation of Ntccrt

Ntccrt is written in C++ and it uses Flext to generate the externals for either Max or Pd, and Gecode [21] for constraint solving and concurrency control. Gecode is an efficient constraint solving library, providing efficient propagators (narrowing operators reducing the set of possible values for some variables). The basic principle of *Ntccrt* is encoding the "when", $\sum$ and "tell" processes as Gecode propagators. The other processes are simulated by storing them into queues for each time-unit.

Although Gecode was designed to solve combinatorial problems, Toro found out in [22] that writing the "when" and the $\sum$ processes as propagators, Gecode can manage all the concurrency needed to represent `ntcc`. Following, we explain the encoding of the "tell" and the "when".

To represent the "tell", we define a super class $Tell$. For *Ntccrt*, we provide three subclasses to represent these processes: **tell** $(a = b)$, **tell** $(a \in B)$, and **tell** $(a > b)$. Other kind of "tells" can be easily defined by inheriting from the $Tell$ superclass and declaring an $Execute$ method.

We have a *When propagator* for the "when" and a *When* class for calling the propagator. A process **when** C **do** P is represented by two propagators: $C \leftrightarrow b$ (a reified propagator for the constraint C) and **if** b **then** P **else** *skip* (the *When propagator*). The *When propagator* checks the value of b. If the value of b is true, it calls the *Execute* method of P. Otherwise, it does not take any action. Figure 2 shows how to encode the process **when** $a = c$ **do** P using our *When propagator*.

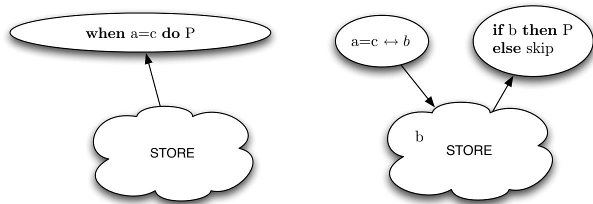

Figure 2. Example of the *When propagator*

5. APPLICATIONS

We selected two real-life applications to show the relevance of using *Ntccrt* externals in Pd. *Ccfomi* shows us how we can use *Ntccrt* to interact in real-time with a human player. Finally, a signal processing application shows us how a *Ntccrt external* can send control signals to trigger signal processing filters.

5.1. Machine Improvisation

Machine improvisation usually consider building representations of music, either by explicit coding of rules or applying machine learning methods.An interactive machine improvisation system capable of real-time perform two activities concurrently: Stylistic learning and Stylistic simulation.

Rueda et al. define in [17], Stylistic learning as the process of applying machine learning methods to musical sequences in order to capture salient musical features and organize these features into a model. On the other hand, Stylistic simulation as the process of producing musical sequences stylistically consistent with the learned material.

A machine improvisation system using `ntcc` is *Ccfomi*. *Ccfomi* executes both phases concurrently, uses `ntcc` to synchronize both phases of the improvisation, and uses the *Factor Oracle* (FO) to store the information of the learned sequences.

FO is a finite state automaton constructed in linear time and space. It has two kind of transitions (*links*). *Factor links* are going forward and following them is possible to recognize at least all the factors from a sequence. *Suffix links* are going backwards and they connect repeated patterns of the sequence. Further formal definitions about FO can be found in [1].

Following, we give a brief description of *Ccfomi* taken from [17]. *Ccfomi* is divided in three subsystems: learning (ADD), improvisation (IMPROV) and playing (PLAYER) running concurrently. In addition, there is a synchronization process (SYNC) in charge of synchronization.

Ccfomi has three kind of variables to represent the partially built FO automaton: Variables $from_k$ are the set of labels of all currently existing *factor links* going forward from k. Variables S_i are *suffix links* from each state i and variable δ_{k,σ_i} give the state reached from k by following a *factor link* labeled σ_i.

In our implementation of *Ccfomi*, the variables $from_k$ and δ_{k,σ_i} are modeled as infinite rational trees [16] with unary branching. That way, we can add new elements to $from$ and δ dynamically.

Rational trees have been subject of multiple researches to construct a constraint system based on them. Using this constraint system is possible to post the constraints $cons(c, nil, B)$, $cons(b, B, C)$, $cons(a, C, D)$ to model a list of three elements $[a, b, c]$.

Following, we explain some *Ccfomi* processes. The ADD process (specified in [17]) is in charge of building the FO by creating the *factor links* and the *suffix links*. This process models the learning phase.

The learning and the simulation phase must work concurrently. In order to achieve that, it is required that the simulation phase only takes place once the subgraph is completely built. The $SYNC$ process is in charge of doing the synchronization between the simulation and the learning phase to preserve that property.

Synchronizing both phases is greatly simplified by the use of constraints. When a variable has no value, the "when" processes depending on it are blocked. Therefore, the $SYNC$ process is "waiting" until go is greater or equal than one. It means that the $PLAYER$ process has played the note i and the ADD process can add a new symbol to the FO. The condition $S_{i-1} \geq 0$ is because the first *suffix link* of the FO is equal to -1 and it cannot be followed in the simulation phase.

$$SYNC_i \stackrel{def}{=} \textbf{when } S_{i-1} \geq -1 \wedge go \geq i \textbf{ do } (ADD_i \parallel \textbf{next } SYNC_{i+1})$$

$\| \textbf{unless } S_{i-1} \geq -1 \wedge go \geq i \textbf{ next } SYNC_i)$

The *PLAYER* (specified in [17]) process simulates a human player. It decides, non-deterministically, each time-unit between playing a note or not. When running this model in Pd, we replace this process by receiving an input (e.g., a MIDI input) from the environment.

The improvisation process *IMPROV* starts from state k and probabilistically, chooses whether to output the symbol σ_k or to follow a backward link S_k. Another probabilistic version of this process can be found in [12].

For this work, we have modeled *IMPROV* as a simpler improvisation process. We are more interested in showing the synchronization between the improvisation phases, than showing how we can control the choice among *suffix links* and *factor links* based on a probabilistic distribution. For that reason, choices in our *IMPROV* process are made non-deterministically.

$$
\begin{aligned}
&IMPROV(k) \stackrel{def}{=} \\
&\quad \textbf{when } S_k = -1 \textbf{ do next} \\
&\quad\quad (\textbf{tell } (out = \sigma_{k+1}) \parallel IMPROV(k+1)) \\
&\quad \parallel \textbf{when } S_k \geq 0 \textbf{ do next} \\
&\quad\quad ((\textbf{tell } (out = \sigma_{k+1}) \parallel IMPROV(k+1)) + \\
&\quad\quad \sum_{\sigma \in \Sigma} \textbf{when } \sigma \in from_{s_k} \textbf{ do} \\
&\quad\quad\quad (\textbf{ tell } (out = \sigma) \parallel IMPROV(\delta_{s_k}, \sigma))) \\
&\quad \parallel \textbf{unless } S_k \geq -1 \textbf{ next } IMPROV(k)
\end{aligned}
$$

The system is modeled as the *PLAYER* and the *SYNC* process running in parallel with a process waiting until n symbols have been played to launch the *IMPROV* process.

$$
\begin{aligned}
SYSTEM_n \stackrel{def}{=} \; & !\textbf{tell}(S_0 = -1) \parallel PLAYER_1 \\
& \parallel SYNC_1 \parallel Wait_n
\end{aligned}
$$

5.2. Signal processing

Ntcc was used in the past as an audio processing framework [18]. In that work, Valencia and Rueda showed how this modeling formalism gives a compact and precise definition of audio stream systems. They argued that it is possible to model an audio system and prove temporal properties using the temporal logic associated to ntcc. They proposed that a ntcc model, where each time-unit can be associated to processing the current sample of a sequential stream.

Unfortunately, in practice it is difficult to implement that model because it will require to execute 44100 time-units per second to process a 44.1 kHz audio stream. This is not possible using our interpreter and using the other ntcc interpreters neither.

Another approach to give formal semantics to audio processing is the visual audio processing language *Faust* [10]. Faust semantics are based on an algebra of block diagrams. This gives a formal and precise meaning to the operation programed there.

Our approach is different, we use a *Ntccrt* external for Pd or Max to synchronize the graphical objects in charge of audio, video or MIDI processing in Pd. For instance, the ntcc external decides when triggering a graphical object in charge of applying a delay filter to an audio stream and it will not allow other graphical objects to apply a filter on that audio stream, until the delay filter finishes its work.

Our system is composed by a collection of n filters and m objects (MIDI, audio or video streams). When a filter P_i is working on an object m_j, another filter cannot work on m_j until P_i is done. A filter P_i is activated when a condition over its input is true. That condition is easily represented by a constraint.

Our system is composed by the infinite rational tree variables *work*, *end* and *input* representing lists. $Work_j$ represents the identifiers of the filter working on the object j. End_j represents when the object j has finished its work. Values for end_j are updated each time-unit with information from the environment. $Input_i$ represents the conditions necessary to launch filter P_i, based on information received from the environment. Finally, $wait_j$ represents the set of filters waiting to work on the object m_j. Note that $work_j$ is a reference to the position j of the list *work* (same with *end* and *input*).

Next, we explain the definitions of our system. Objects are represented by *IdleObject* and *BusyObject*. An object is *idle* until it non-deterministically chooses a filter from the $wait_j$ variable. After that, it will remain *busy* until the $end_j = true$ constraint can be deduced from the store.

$$
\begin{aligned}
&IdleObject(j) \stackrel{def}{=} \\
&\quad \textbf{when } work_j > 0 \textbf{ do next } BusyObject(j) \\
&\quad \parallel \textbf{unless } work_j > 0 \textbf{ next } IdleObject(j) \\
&\quad \parallel \sum_{x \in P} \textbf{when } x \in wait_j \textbf{ do tell } work_j = x
\end{aligned}
$$

$$
\begin{aligned}
&BusyObject(j) \stackrel{def}{=} \\
&\quad \textbf{when } end_j = true \textbf{ do } IdleObject(j) \\
&\quad \parallel \textbf{unless } end_j = true \textbf{ next } BusyObject(j)
\end{aligned}
$$

Filters are represented by the definitions *IdleFilter*, *WaitingFilter* and *BusyFilter*. A filter is *idle* until it can deduce that $input_i = true$. $Input_i$ can be a condition based on multiple criteria.

$$
\begin{aligned}
&IdleFilter(i, j) \stackrel{def}{=} \\
&\quad \textbf{when } input_i = true \textbf{ do } WaitFilter(i, j) \\
&\quad \parallel \textbf{unless } input_i = true \textbf{ next } IdleFilter(i, j)
\end{aligned}
$$

A filter is *waiting* when the information for launching it can be deduced from the store, but it has not yet control over the object m_j. When it can control the object, it calls the definition *BusyFilter*.

$$
\begin{aligned}
&WaitingFilter(i,j) \stackrel{def}{=} \\
&\quad \textbf{when } work_j = i \textbf{ do } BusyFilter(i,j) \\
&\parallel \textbf{unless } work_j = i \textbf{ next} \\
&\qquad WaitingFilter(i,j) \parallel \textbf{tell } i \in wait_j
\end{aligned}
$$

A filter is *busy* until it can deduce that the filter finished working on the object associated to it.

$$
\begin{aligned}
&BusyFilter(i,j) \stackrel{def}{=} \\
&\quad \textbf{when } end_j = true \textbf{ do } IdleFilter(i,j) \\
&\parallel \textbf{unless } end_j = true \textbf{ next } BusyFilter(i,j)
\end{aligned}
$$

Filter definitions can be written in OpenMusic using a graphical "patch" (fig 3).

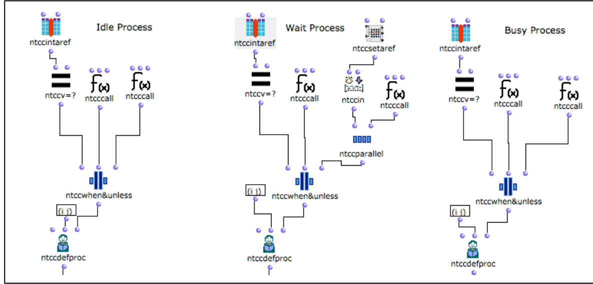

Figure 3. Specifying a *Ntccrt* external in OpenMusic.

The following definition models a situation with two objects and four filters. The external generated for this model can control all kind of objects and filters, represented by graphical objects in Pd.

$$
\begin{aligned}
&System() \stackrel{def}{=} \\
&\quad IdleObject(1) \parallel IdleObject(2) \parallel IdleFilter(1,1) \\
&\parallel IdleFilter(1,2) \parallel IdleFilter(2,1) \parallel IdleFilter(2,2)
\end{aligned}
$$

6. RESULTS

We ran *Ccfomi* as an stand-alone application over an Intel 2.8 GHz iMac using Mac OS 10.5.2 and Gecode 2.2.0. Each time-unit took an average of 20 ms, scheduling around 880 processes per time-unit. We simulated 300 time-units and we ran each simulation 100 times in our tests.

Pachet argues in [11] that an improvisation system able to learn and produce sequences in less than 30ms is appropriate for real-time interaction. Since our implementation of *Ccfomi* has a response time of 20ms in average, we conclude that it is capable of real-time interaction for a 300 (or less) time-units simulation.

For this work, we made all the test under Mac OS X using Pd. Since we are using Gecode and Flext to generate the externals, they could be easily compiled to other platforms and for Max. This is due to Gecode and Flext portability.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we present *Ntccrt* as a framework to manage concurrency in Max and Pd. In addition, we present two real-life applications, a machine improvisation system and a signal processing system. We ran both applications creating *Ntccrt* external objects for Pd.

We want to encourage the use of process calculi to develop reactive systems. For that reason, this research focuses on developing real-life applications with `ntcc` and showing that our interpreter *Ntccrt* is a user-friend-ly tool, providing a graphical interface to specify `ntcc` models and compiling them to efficient C++ programs capable of real-time interaction in Pd.

We argue that using process calculi (such as `ntcc`) to model, verify and execute reactive systems decreases the development time and guarantees correct process synchronization, in contrast to the graphical patch para-digm of Max or Pd. We argue that using that paradigm is difficult and time-demanding to synchronize processes depending on complex conditions. On the other hand, using Ntccrt, we can model such systems with a few graphical boxes in OpenMusic or with a few lines in Common Lisp, representing complex conditions by constraints.

One may argue that although we can synchronize *Ntccrt* with an external clock (e.g., a metronome object) provided by Max or Pd, this does not solve the problem of simulating models when the clock step is shorter than the time necessary to compute a time-unit. To solve this problem, Sarria proposed to develop an interpreter for the Real Time Concurrent Constraint (`rtcc` [20]) calculus, which is an extension of `ntcc` capable of modeling *time-units* with fixed duration.

One may also argue that we encourage formal verification for `ntcc`, but there is not an existing tool to verify these models automatically, not even semi-automatically. To solve this problem, Pérez and Rueda proposed to develop a verification tool for the Probabilistic Timed Concurrent Constraint (`pntcc` [12]) calculus. Currently, they are able to generate an input for Prism [7] based on a `pntcc` model.

In the future, we would like to explore the ideas proposed by Sarria, Pérez and Rueda. Moreover, we want to extend our implementation to support `pntcc` and `rtcc`, and to generate an input for Spin [6] based on a `ntcc` model.

8. ACKNOWLEDGMENTS

We want to thank to Arshia Cont for giving us this idea of using *Ntccrt* in Pd and Max; Fivos Maniatakos, Jorge Pérez and Carlos Toro-Bermúdez for their valuable reviews on this

paper; and Jean Bresson, Gustavo Gutiérrez, and the Gecode developers for their help during the development of *Ntccrt*.

9. REFERENCES

[1] C. Allauzen, M. Crochemore, and M. Raffinot, "Factor oracle: A new structure for pattern matching," in *Conference on Current Trends in Theory and Practice of Informatics*, 1999, pp. 295–310.

[2] A. Allomber, G. Assayag, M. Desainte-Catherine, and C. Rueda, "Concurrent constraint models for interactive scores," in *Proc. of the 3rd Sound and Music Computing Conference (SMC), GMEM, Marseille*, may 2006.

[3] AVISPA. (2008) Ntccsim: A simulation tool for timed concurrent processes. [Online]. Available: http://cic.puj.edu.co/wiki/doku.php?id=grupos:avispa:ntccsim.

[4] J. Bresson, C. Agon, and G. Assayag, "Openmusic 5: A cross-platform release of the computer-assisted composition environment," in *10th Brazilian Symposium on Computer Music*, Belo Horizonte, MG, Brésil, 2005.

[5] S. L. D. Fober, Y. Orlarey, *Midishare: une architecture logicielle pour la musique*. Hermes, 2004, pp. 175–194.

[6] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley Professional, September 2003.

[7] M. Kwiatkowska, G. Norman, D. Parker, and J. Sproston, *Modeling and Verification of Real-Time Systems: Formalisms and Software Tools*. John Wiley & Sons, 2008, ch. Verification of Real-Time Probabilistic Systems, pp. 249–288.

[8] P. Muñoz and A. Hurtado, "Programming robot devices with a timed concurrent constraint programming," in *Principles and Practice of Constraint Programming - CP2004. LNCS 3258, page 803.Springer*, 2004.

[9] M. Nielsen, C. Palamidessi, and F. Valencia, "Temporal concurrent constraint programming: Denotation, logic and applications," *Nordic Journal of Computing*, vol. 1, 2002.

[10] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and semantical aspects of faust," *Soft Comput.*, vol. 8, no. 9, pp. 623–632, 2004.

[11] F. Pachet, "Playing with virtual musicians: the continuator in practice," *IEEE Multimedia*, vol. 9, pp. 77–82, 2002.

[12] J. Pérez and C. Rueda, "Non-determinism and probabilities in timed concurrent constraint programming," in *ICLP*, ser. Lecture Notes in Computer Science, vol. 5366. Springer, 2008, pp. 677–681.

[13] M. Puckette, "Pure data," in *Proceedings of the International Computer Music Conference. San Francisco 1996*, 1996.

[14] M. Puckette, T. Apel, and D. Zicarelli, "Real-time audio analysis tools for Pd and MSP," in *Proceedings of the International Computer Music Conference.*, 1998.

[15] L. Quesada, C. Rueda, , and G. Tamura, "The visual model of cordial," in *Proceedings of the CLEI97. Valparaiso, Chile.*, 1997.

[16] V. Ramachandran and P. V. Hentenryck, "Incremental algorithms for constraint solving and entailment over rational trees," in *Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*. London, UK: Springer-Verlag, 1993, pp. 205–217.

[17] C. Rueda, G. Assayag, and S. Dubnov, "A concurrent constraints factor oracle model for music improvisation," in *XXXII CLEI 2006*, 2006.

[18] C. Rueda and F. Valencia, "A temporal concurrent constraint calculus as an audio processing framework," in *SMC 05*, 2005.

[19] V. A. Saraswat, *Concurrent Constraint Programming*. MIT Press, 1992.

[20] G. Sarria, "Formal models of timed musical processes," Ph.D. dissertation, Universidad del Valle, Colombia, 2008.

[21] C. Schulte and P. J. Stuckey, "Efficient constraint propagation engines," *CoRR*, vol. abs/cs/0611009, 2006.

[22] M. Toro-Bermúdez, "Exploring the possibilities and limitations of concurrent programming for multimedia interaction and graphical representations to solve musical csp's," IRCAM, Paris, Tech. Rep. 2008-3, 2008.