Journal of Theoretical and Applied Information Technology
10th December 2015. Vol.82. No.1
© 2005 - 2015 JATIT & LLS. All rights reserved·
ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

NTCCRT: A CONCURRENT CONSTRAINT FRAMEWORK FOR SOFT REAL-TIME MUSIC INTERACTION

[1]MAURICIO TORO, [2]CAMILO RUEDA, [3]CARLOS AGÓN, [4]GÉRARD ASSAYAG
[1]Asstt Prof., Department of Informatics and Systems, UNIVERSIDAD EAFIT, Colombia
[2]Prof., Department of Computer Science, PONTIFICIA UNIVERSIDAD JAVERIANA CALI, Colombia
[3]Prof., Music Modeling Team, IRCAM, UMR 9912 CNRS, France
[4]Dr., Music Modeling Team, IRCAM, UMR 9912 CNRS, France
E-mail: [1]mtorobe@eafit.edu.co, [2]crueda@cic.puj.edu.co, [3]assayag@ircam.fr, [4]carlos.agon@ircam.fr

ABSTRACT

Writing music interaction systems is not easy because their concurrent processes usually access shared resources in a non-deterministic order, often leading to unpredictable behavior. Using Pure Data (Pure Data) and Max/MSP, it 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, declaratively, the synchronization of multiple criteria as constraints. In this article, we propose the framework Ntccrt, as a new alternative to manage concurrency in Pure Data and Max/MSP. Ntccrt is a real-time capable interpreter for ntcc. Using Ntccrt binary plugins in Pure Data, we executed models for machine improvisation and signal processing. We also analyzed two case studies: one of a machine improvisation system and one of a signal processing system. We found out that performance of both case studies is compatible with soft real-time music interaction; it means, a musician can interact with Ntccrt without noticeable delays during the interaction.

Keywords: *Concurrent Constraint Programming (ccp), Soft Real-Time, Machine Improvisation, Signal Processing, Music Interaction, Computer Music, Process Calculi.*

1. INTRODUCTION

Music 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. Process calculi has been applied to the modeling of ecological systems [16, 39, 17, 38] and interactive music systems [12, 4, 29, 27]. As an example, using the process calculus *non-deterministic timed concurrent constraint* (ntcc) [11], we can model reactive systems with synchronous, asynchronous or non-deterministic behavior. Ntcc and its extensions have been used to model interactive systems such as an audio processing [23, 36], machine improvisation [22, 28, 15, 25, 33], and interactive scores [2, 3, 25, 37, 31,30, 32, 34, 35].

Although there are three interpreters to simulate the execution of ntcc, they are not suitable for soft real-time music interaction. It means that they are not able to interact with a musician without letting the musician experience noticeable delays in the interaction with the computer program.

We can also program soft real-time systems for music interaction and signal processing using C++. Unfortunately, C++ requires long development time. To overcome that problem, programming languages such as Pure Data [18] and Cycling 74's Max/MSP [19], provide a visual programming paradigm to program soft real-time systems and they include several programming interfaces for concurrent programming.

1.1 The problem

It is a well-known problem that it is not possible to implement process synchronization of concurrent processes written in Pure Data and Max/MSP using a declarative approach. Although Pure Data and Max/MSP support concurrency, it is a hard task to trigger or halt the execution of a process based on multiple criteria. As an example, using Pure Data or Max/MSP, 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 visual programming 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) has played the note F# among the last four notes.

1.2 Solution using Ntccrt

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" ($G_7 \in \text{PlayedChordsSet}$).

In this article, we propose using ntcc to manage concurrency in Pure Data and Max/MSP, executing ntcc models on *Ntccrt*[1]. On *Ntccrt*, ntcc models can be compiled as an *binary plugin* for Pure Data or Max/MSP. Additionally, the binary plugins can be specified, textually, using Common Lisp or using visual programming in OpenMusic [6]. We recommend the use of OpenMusic. We argue that concurrent visual programming, usually based on process calculi such as Cordial [20], makes the power of concurrency available for a wider range of users.

1.3 Contributions of this article

Our framework *Ntccrt*[2] is composed by the following components: (1) The ntcc interpreter written in C++, (2) interfaces for both Common Lisp and OpenMusic, and (3) the implementation of two case studies.

1.4 Structure of the article

The remainder of this article is structured as follows. Section 2, intuitively, explains the semantics of ntcc processes and gives some examples of simple ntcc processes modeling music interaction. Section 3 explains related work on ntcc interpreters and threading programming libraries available for Pure Data and Max/MSP. Section 4, discusses two case studies of *Ntccrt* to model a music interaction and a signal processing system. Section 5 explains the simulation results of the case studies. Finally, Section 6 gives concluding remarks, states limitations of this approach and proposes future works.

2. THE NTCC CALCULUS

A family of process calculi is *concurrent constraint programming* (ccp) [24], where a system is modeled in terms of variables and constraints over some variables. Furthermore, there are processes 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 □) 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*. For instance, using a finite-domain constraint system we can deduce the constraint $pitch \neq 60$ from the constraints $pitch > 40$ and $pitch < 59$.

We can choose an appropriate constraint system to model any problem; however, 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 the 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 each time we change the value of *x*. As an example, 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 $pitch_3 = G$".

In what follows, we give some examples of how the computational processes of ntcc can be used with a FD constraint system. A summary can be

1 This research was partially founded by the REACT project, sponsored by Colciencias.
2 http://ntccrt.sourceforge.net

found in Table 1. The semantics of `ntcc` can be found at [11].

- Using the "tell", it is possible to add constraints such as **tell** (*pitch* = *60*), meaning that must be equal to 60 or **tell** (59 < *pitch* < 10*1*), meaning that *pitch* 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* = *60*) ‖ **when** *pitch* = *60* **do tell** (*Instrument* = 1) is a process telling the store that is 62 and concurrently assigning the instrument to one, since *pitch* is in desired octave.
- The "next" is useful when we want to model variables changing through time; for instance, **when** *pitch* = *60* **do next tell** (*pitch* ≠ *60*) , means that if 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* = *60* **next tell** (*lastpitch* ≠ 60) reacts when is false or when cannot be deduced from the store (e.g., 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 is used to model non-deterministic choices. For instance, $!\sum_{i \in 48,52,55}$.**when** i □ *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 $q(x) \overset{\text{def}}{=} P_q$, where *q* is the process name and 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.

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: Summary of `ntcc` *processes (a.k.a. agents)*

3. RELATED WORK

In this section, we present related work about concurrency support for Pure Data and Max/MSP, and available interpreters for `ntcc`.

3.1. Concurrency in Pure Data and Max/ MSP

To program concurrent programs on Max/MSP and Pure Data, we can use their message passing programming libraries. We can also create binary plugins in C++. In fact, we can use any existing threading programming library for C++ to write binary plugins for both, Pure Data and Max/MSP. There is also a native programming library for Max/MSP 7. Another way to write an binary plugin is using the *Flext library*[2]. Flext provides a unique interface to write, in the C++ language, binary plugins dealing with both, Pure Data and Max/MSP.

3.2 Ntcc interpreters

There are three interpreters available for `ntcc`: *Lman* [10] used as a framework to program LegoTM robots, *NtccSim* [5] used to model and verify properties of biological systems, and *Rueda's interpreter* [22] for music interaction.

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

2 http://grrrr.org/research/software/flext/

interpreter [22]. Both, *Lman* and *Rueda's interpreter* executed the model giving the expected output; However, they were not capable of executing music interaction systems in soft real-time.

4. THE NTCCRT FRAMEWORK

Ntccrt is a framework to specify and execute ntcc models capable of soft real-time music interaction.

4.1. Design of Ntccrt

The 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 binary plugin object for Pure Data or Max/MSP.

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

To write a `ntcc` model in *Ntccrt*, the users may write them directly in C++, using a parser that takes Common Lisp macros or the use may build a program in OpenMusic. Using either of these representations, it is possible to generate a stand-alone program or an binary plugin as shown in Figure *1*.

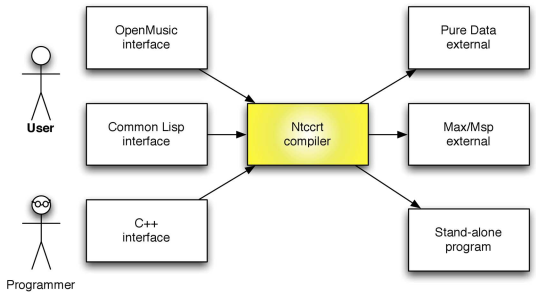

Figure 1: Developing mode of Ntccrt.

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 binary plugin for either Pure Data or Max/MSP. An advantage of using the binary plugins lies on using control signals and the message passing programming library provided by Pure Data and Max/MSP to synchronize any *object* with the *Ntccrt* binary plugin.

To handle *musical instrument digital interface* (MIDI) streams we use the predefined functions in Pure Data or Max/MSP to process MIDI. Then, we connect the output of those functions to the *Ntccrt* binary plugin. We also provide an interface for *Midishare* [7], useful when running stand-alone programs.

4.2 Implementation of Ntccrt

Ntccrt is written in C++ and it uses Flext to generate the binary plugins for either Max/MSP or Pure Data, and Gecode [26] 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", 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 [27] that writing the "when" and the processes as propagators, Gecode can manage all the concurrency needed to represent ntcc.

In what follows, we explain the encoding of the "tell" and the "when" processes.

- 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 □ B), and **tell** ($a > b$). Other kind of "tells" can be easily defined by inheriting from the *Tell* superclass and declaring an *execute* method that calls the propagator for the constraint (e.g., a = b or *a* □ *B).*
- To represent the "when", we define a class *When*. The class *When* calls 2 propagators. 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 the *when propagator*.

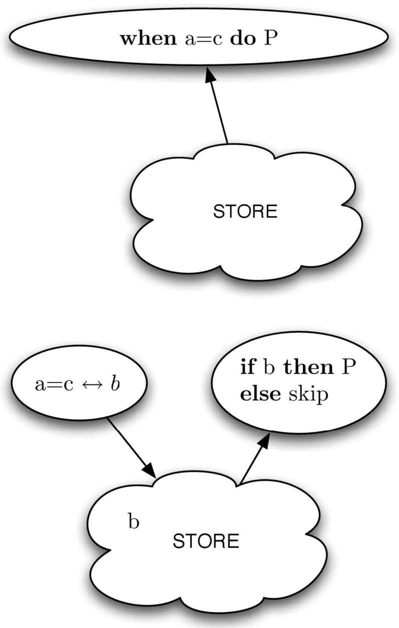

Figure 2: Example of the when process propagator

5. CASE STUDIES

We selected two case studies to show the relevance of using *Ntccrt* binary plugins in Pure Data. First, the machine improvisation system *Ccfomi* shows us how we can use *Ntccrt* to interact in real-time with a human player. Second, a signal processing application shows us how a *Ntccrt binary plugin* can send control signals to trigger signal processing filters.

5.1 Machine Improvisation

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

Rueda et al. define in [22], 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 and stylistic simulation as the process of producing musical sequences stylistically consistent with the learned material.

A machine improvisation system using `ntcc` is "A Concurrent Constraint Factor Oracle Model for Music Improvisation" (*ccfomi*). *Ccfomi* executes both phases concurrently, it uses `ntcc` to synchronize both phases of the improvisation, and uses the *factor oracle* to store the information of the learned sequences.

The factor oracle 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 *factor oracle* can be found in [1].

Following, we give a brief description of *ccfomi* taken from [22]. *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 3 kind of variables to represent the partially built *factor oracle* 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 $\delta_{k,\sigma i}$ gives the state reached from *k* by following a *factor link* labeled σ_i.

In our implementation of *ccfomi*, the variables and are modeled as infinite rational trees [21] with unary branching. That way, we can add new elements to $from_k$ and $\delta_{k,\sigma i}$ 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*].

In what follows, we explain some *ccfomi* processes. The *ADD* process (specified in [22]) 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 *factor oracle*. The condition $S_{i-1} > 0$ is because the first *suffix link* of the factor oracle is equal to -1 and it cannot be followed in the simulation phase.

$$
\begin{aligned}
&SYNC_i \overset{def}{=} \\
&\quad \textbf{when } S_{i-1} \geq -1 \land go \geq i \textbf{ do} \\
&\qquad (ADD_i \parallel \textbf{next } SYNC_{i+1}) \\
&\quad \parallel \textbf{unless } S_{i-1} \geq -1 \land go \geq i \textbf{ next } SYNC_i)
\end{aligned}
$$

The *PLAYER* (specified in [22]) process simulates a human player. It decides, non-deterministically, each time-unit between playing a note or not. When running this model in Pure Data, 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. A probabilistic version of this process can be found in [15]. For this work, we have modeled *IMPROV* as a simpler improvisation process than the model in [15]. 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 the *IMPROV* process are made non-deterministically.

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

The machine improvisation 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.

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

5.2 Signal processing

`Ntcc` was used in the past as an audio processing framework [23]. 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 Ntccrt nor using the other `ntcc` interpreters neither.

Another approach to give formal semantics to audio processing is the visual audio processing language *Faust* [13]. Faust semantics are based on an algebra of block diagrams. This gives a formal and precise meaning to the operation.

Our approach is different from Faust's [13] and Rueda and Valencia's [23], we use a *Ntccrt* binary plugin for Pure Data or Max/MSP to synchronize objects in charge of audio, video or MIDI processing in Pure Data; for instance, the `ntcc` binary plugin decides when triggering an object in charge of applying a delay filter to an audio stream and it will not allow other 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 is working on an object , another filter cannot work on until is done. A filter 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. $Input_j$ represents the conditions necessary to launch filter P_j, based on information received from the environment. Finally, $wait_j$ represents the set of filters waiting to work on the object . Note that $work_j$ is a reference to the position *j* of the list *work* (same with *end* and *input*).

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

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

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

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

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

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*.

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

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

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

Filter definitions can be written in OpenMusic using the visual programming paradigm as shown in Figure *3*.

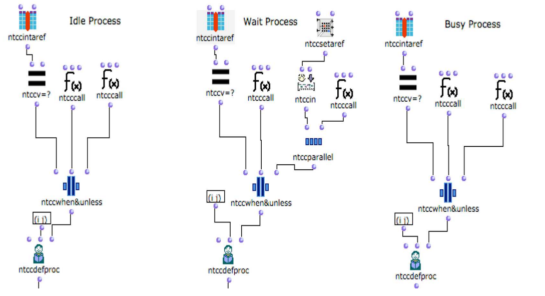

Figure 3: Using a Ntccrt binary plugin in OpenMusic.

The following definition models a situation with two objects and four filters. The binary plugin generated for this model can control all kind of objects and filters, represented by objects in Pure Data.

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

6. RESULTS

We executed *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 `ntcc` processes per time-unit. We simulated 300 time-units and we executed each simulation 100 times in the tests.

Pachet argues in [14] that an improvisation system able to learn and produce sequences in less than 30ms is appropriate for soft real-time music 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 Pure Data. Since we are using Gecode and Flext to generate the binary plugins, they could be easily compiled to other platforms and for Max/MSP. This is due to Gecode and Flext portability.

7 . CONCLUSIONS AND FUTURE WORK

We recall from the Introduction that although Pure Data and Max/MSP support concurrency, it is a hard task to trigger or halt the execution of a process based on multiple criteria.

In this article, we introduce *Ntccrt* as a framework to manage concurrency in Max/MSP and Pure Data. In addition, we present two case studies, a machine improvisation system and a signal processing system. We executed both case studies creating *Ntccrt* binary plugins for Pure Data.

We want to encourage the use of process calculi to develop reactive systems. For that reason, this research focuses on developing real-life case studies with ntcc and showing that our interpreter *Ntccrt* is a user-friendly tool, providing a visual programming interface to specify ntcc models and compiling them to efficient C++ programs capable of real-time interaction in Pure Data.

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 visual programming paradigm of Max/MSP or Pure Data. We argue that using that paradigm is difficult and time-demanding to synchronize processes depending on complex conditions. 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.

7.1 Future work

One may argue that although we can synchronize *Ntccrt* with an binary plugin clock (e.g., a metronome object) provided by Max/MSP or Pure Data, 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 [25]) 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 [15]) calculus. Currently, they are able to generate an input for Prism [9] 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 [8], based on a ntcc model, for model checking.

7.2 Limitations of Ntccrt

There is a limitation of Ntccrt. It is difficult to implement a signal processing 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 Ntccrt nor using other ntcc interpreters neither. For this reason, we propose using Ntccrt as a framework only to control signal processing operation programmed in Max/MSP, Pure Data or C++ and not as signal processing framerwork itself.

ACKNOWLEDGMENTS

We want to thank to Arshia Cont for giving us this idea of using *Ntccrt* in Pure Data and Max; Fivos Maniatakos, Jorge Pérez and Carlos Toro-Bermúdez for their valuable reviews on this article; and Jean Bresson, Gustavo Gutiérrez, and the Gecode developers for their help during the development of *Ntccrt*.

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*, pages 295–310, 1999.

[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] A. Allombert, M. Desainte-Catherine, and M. Toro. Modeling temporal constrains for a system of interactive score. In G. Assayag and C. Truchet, editors, *Constraint Programming in Music*, chapter 1, pages 1–23. Wiley, 2011.

[4] J. Aranda, G. Assayag, C. Olarte, J. A. Pérez, C. Rueda, M. Toro, and F. D. Valencia. An overview of FORCES: an INRIA project on declarative formalisms for emergent systems. In P. M. Hill and D. S. Warren, editors, *Logic Programming, 25th International Conference, ICLP 2009, Pasadena, CA, USA, July 14-17, 2009. Proceedings*, volume 5649 of *Lecture Notes in Computer Science*, pages 509–513. Springer, 2009.

[5] AVISPA. Ntccsim: A simulation tool for timed concurrent processes, 2008. Available at http://cic.puj.edu.co/~jg/ntccSim/

[6] 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.

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

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

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

[10] 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.

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

[12] C. Olarte, C. Rueda, G. Sarria, M. Toro, and F. Valencia. Concurrent Constraints Models of Music Interaction. In G. Assayag and C. Truchet, editors, *Constraint Programming in Music*, chapter 6, pages 133–153. Wiley, Hoboken, NJ, USA., 2011.

[13] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of faust. *Soft Comput.*, 8(9):623–632, 2004.

[14] F. Pachet. Playing with virtual musicians: the continuator in practice. *IEEE music*, 9:77–82, 2002.

[15] J. Pérez and C. Rueda. Non-determinism and probabilities in timed concurrent constraint programming. In *ICLP*, volume 5366 of *Lecture Notes in Computer Science*, pages 677–681. Springer, 2008.

[16] A. Philippou and M. Toro. Process Ordering in a Process Calculus for Spatially-Explicit Ecological Models. In *Proceedings of MOKMASD'13*, LNCS 8368, pages 345–361. Springer, 2013.

[17] A. Philippou, M. Toro, and M. Antonaki. Simulation and Verification for a Process Calculus for Spatially-Explicit Ecological Models. *Scientific Annals of Computer Science*, 23(1):119–167, 2013.

[18] M. Puckette. Pure data. In *Proceedings of the International Computer Music Conference. San Francisco 1996*, 1996.

[19] M. Puckette, T. Apel, and D. Zicarelli. Real-time audio analysis tools for Pure Data and MSP. In *Proceedings of the International Computer Music Conference.*, 1998.

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

[21] 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*, pages 205–217, London, UK, 1993. Springer-Verlag.

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

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

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

[25] G. Sarria. *Formal Models of Timed Musical Processes*. PhD thesis, Universidad del Valle, Colombia, 2008.

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

[27] M. Toro. Exploring the possibilities and limitations of concurrent programming for music interaction and graphical representations to solve musical csp's. Technical Report 2008-3, Ircam, Paris (FRANCE), 2008.

[28] M. Toro. Probabilistic Extension to the Factor Oracle Model for Music Improvisation. Master's thesis, Pontificia Universidad Javeriana Cali, Colombia, 2009.

[29] M. Toro. Towards a correct and efficient implementation of simulation and verification tools for probabilistic ntcc. Technical report, Pontificia Universidad Javeriana, May 2009.

[30] M. Toro. Structured interactive musical scores. In M. V. Hermenegildo and T. Schaub, editors, *Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK*, volume 7 of *LIPIcs*, pages 300–302. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2010.

[31] M. Toro. *Structured Interactive Scores: From a simple structural description of a music scenario to a real-time capable implementation with formal semantics* . PhD thesis, Univeristé de Bordeaux 1, France, 2012.

[32] M. Toro. Structured interactive music scores. *CoRR*, abs/1508.05559, 2015.

[33] M. Toro, C. Agón, G. Assayag, and C. Rueda. Ntccrt: A concurrent constraint framework for real-time interaction. In *Proc. of ICMC '09*, Montreal, Canada, 2009.

[34] M. Toro and M. Desainte-Catherine. Concurrent constraint conditional branching interactive scores. In *Proc. of SMC '10*, Barcelona, Spain, 2010.

[35] M. Toro, M. Desainte-Catherine, and P. Baltazar. A model for interactive scores with temporal constraints and conditional branching. In *Proc. of Journées d'Informatique Musical (JIM) '10*, May 2010.

[36] M. Toro, M. Desainte-Catherine, and J. Castet. An extension of interactive scores for music scenarios with temporal relations for micro and macro controls. In *Proc. of Sound and Music Computing (SMC) '12*, Copenhagen, Denmark, July 2012.

[37] M. Toro, M. Desainte-Catherine, and C. Rueda. Formal semantics for interactive music scores: a framework to design, specify properties and execute interactive scenarios. *Journal of Mathematics and Music*, 8(1):93–112, 2014.

[38] M. Toro, A. Philippou, S. Arboleda, C. Vélez, and M. Puerta. Mean-field semantics for a Process Calculus for Spatially-Explicit Ecological Models. Technical report, Department of Informatics and Systems, Universidad Eafit, 2015. Available at http://blogs.eafit.edu.co/giditic-software/2015/10/01/mean-field/.

[39] M. Toro, A. Philippou, C. Kassara, and S. Sfenthourakis. Synchronous parallel composition in a process calculus for ecological models. In G. Ciobanu and D. Méry, editors, *Proceedings of the 11th International Colloquium on Theoretical Aspects of Computing - ICTAC 2014, Bucharest, Romania, September 17-19*, volume 8687 of *Lecture Notes in Computer Science*, pages 424–441. Springer, 2014.