NTCCRT: A CONCURRENT CONSTRAINT FRAMEWORK FOR REAL-TIME INTERACTION

Mauricio Toro-Bermúdez, Camilo Rueda
Universidad Javeriana Cali (PUJC)

Carlos Agón
IRCAM

Gérard Assayag
IRCAM - CNRS UMR 9912

ABSTRACT

Multimedia interaction systems are complex. Concurrent processes may access shared resources in a non-deterministic order, leading to unpredictable behavior. Using Pure Data (Pd) and Max/MSP is possible to program concurrency, but it is difficult to synchronize processes based on multiple criteria. The Non-deterministic Timed Concurrent Constraint (ntcc) calculus overcome that problem by representing multiple criteria as constraints. Ntccrt is our real-time capable interpreter for ntcc. Ntccrt can manage concurrency in Pd and Max. Using Ntccrt binary plugins in Pd, we ran models for machine improvisation and signal processing.

1. INTRODUCTION

Multimedia interaction systems—inhomogeneously 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 those systems.

For instance, using ntcc [3], we can model reactive systems. Ntcc has been used to model audio processing [7] and a real-time (RT) machine improvisation [6].

There are three interpreters for ntcc, but they are not suitable for RT interaction because they are not able to interact with the users without letting them experience noticeable delays in the interaction.

On the other hand, we can program RT systems using C++. Unfortunately, using C++ requires long development time. To overcome that problem, programming languages such as Pure Data (Pd) and Max/MSP [5], provide a graphical interface to program RT systems and use APIs for concurrent programming (e.g., Max 5 SDK, Flext and Pthreads).

Although Pd and Max support concurrency, it is a hard task to trigger or halt 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) has played the chord G7, and (2) played the note F# among the last four.

Using ntcc (described in section 2), we can represent the complex condition C presented above as the conjunction of constraints \((c_1 \land c_2)\). Each constraint (i.e., mathematical condition) represents a criterion declaratively. For instance, condition (1) is represented by the constraint "G7 is on the set of played chords" \((G_7 \in \text{PlayedChords})\).

Since it is difficult to program concurrency in Max and Pd, we propose executing ntcc models on our framework Ntccrt1. 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 (OM) [2]. Using OM makes the power of concurrency available for a wider range of users.

In what follows, we present Ntccrt\(^2\) in section 3. In addition, we explain the implementation of two real-life applications in section 4, performance results on section 5, and concluding remarks and future works in section 6.

2. THE NTCC CALCULUS

In ntcc, a system is modeled in terms of variables and constraints over some variables. There are also agents reasoning about partial information (by the means of constraints) about the system variables contained on a common store.

Ntcc is based on the idea of a constraint system (CS). A CS 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 ntcc system may include several CSs for different variable types. A CS providing arithmetic relations over natural numbers is known as Finite Domain (FD). Using a FD CS we can deduce the constraint \(\text{pitch} \neq 6\) from the constraint \(\text{pitch} < 4\).

Ntcc also provides a 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.

Following, we give some examples of how the computational agents of ntcc can be used with a FD constraint system. Table 1 presents the agents used in our applications.

---

1This research was partially founded by Colciencias-REACT project.
2Sources and binaries can be found at http://ntccrt.sourceforge.net
The “when” can be used to describe how the system reacts to different events. For instance, \( \text{when } \text{pitch}_1 = 60 \) \( \text{do } \text{tell} \) \( \text{CMayor} = \text{true} \) is a process reacting as soon as the pitch sequence \( C, E, G \) has been played, adding the constraint \( \text{CMayor} = \text{true} \) to the store (with a “tell” agent) in the current time-unit.

Using “next” we can model variables changing through time. For instance, \( \text{when } (\text{pitch}_1 = 60) \text{ do next } \text{tell} \) \( (\text{pitch}_1 \neq 60) \), means that if \( \text{pitch}_1 \) 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, executing a process in the next time-unit.

The \( \Sigma \) is used to model non-deterministic choices. For instance, \( \Sigma_{\forall i \in P} \text{when } (\text{pitch}_i = i) \text{ do tell} \) \( \text{(pitch } = i) \) models a system where each time-unit, it chooses a note among the notes played previously that belongs to the C major chord (represented by pitches 48, 52, 55).

Finally, a basic recursion is defined in \text{ntcc} with the form \( q(x) \overset{\text{def}}{=} P_q, \) 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”. Recursion is used to model iteration and recursive definitions.

<table>
<thead>
<tr>
<th>Agent</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{tell} ( (c) )</td>
<td>Adds ( c ) to the current store</td>
</tr>
<tr>
<td>\text{when} ( (c) \text{ do } A ) ( A \parallel B )</td>
<td>Parallel composition</td>
</tr>
<tr>
<td>\text{next } A</td>
<td>Runs ( A ) at the next time-unit</td>
</tr>
<tr>
<td>\text{unless} ( (c) \text{ next } A )</td>
<td>Unless ( c ) can be inferred now, run ( A )</td>
</tr>
<tr>
<td>( \Sigma_{\forall i \in \text{PlayedPitches}} \text{when } (\text{pitch}_i = i) \text{ do tell} )</td>
<td>Chooses ( P_i ) s.t. ( (\text{pitch}_i) ) holds</td>
</tr>
<tr>
<td>!P</td>
<td>Executes ( P ) each time-unit</td>
</tr>
</tbody>
</table>

Table 1. Some \text{ntcc} agents.

3. OUR FRAMEWORK: NTCCRT

\text{Ntccrt} is our framework to execute \text{ntcc} models. \text{Ntccrt} works on two modes, one for writing the models and another one for executing those models.

To write a \text{ntcc} model in \text{Ntccrt}, the users may write it directly in C++, using Common Lisp macros or writing a graphical “patch” in OpenMusic.

To execute a \text{Ntccrt} program, the users can create a stand-alone program or they can create an external for either Pd or Max. With the \text{Ntccrt} externals, the users can use control signals and the message passing API, provided by both Pd and Max, to synchronize any Max or Pd object.

\text{Ntccrt} is written in C++ and it uses Flext to generate the externals for either Max or Pd, and Gecode [8] 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 \text{Ntccrt} is encoding the “when”, \( \Sigma \) and “tell” processes as Gecode propagators. The other processes are simulated by storing them into queues representing each time-unit.

Although Gecode was designed to solve combinatorial problems, Toro found out in [9] that writing the “when”, \( \Sigma \) and “tell” processes as propagators, Gecode can manage all the concurrency required to represent \text{ntcc}.

4. APPLICATIONS

We present two applications ran using \text{Ntccrt} externals.

4.1. Machine Improvisation

Machine improvisation sometimes consider building representations of music by applying machine learning methods.

A machine improvisation system capable of real-time performs two activities concurrently: the process of applying machine learning methods to musical sequences in order to capture salient musical features and the process of producing musical sequences stylistically consistent with the learned material.

A machine improvisation system using \text{ntcc} is “A Concurrent Constraint Factor Oracle Model for Music Improvisation” \text{Ccfomi} [6]. It uses \text{ntcc} to synchronize both phases of the improvisation, and it uses the \text{Factor Oracle (FO)} [1] to store the information of the learned sequences.

The \text{FO} is a finite state automaton built efficiently. It has two kind of transitions (links). \text{Factor links} are going forward and following them one recognizes at least all the factors from a sequence. \text{Suffix links} are going backwards and they connect repeated patterns of the sequence.

Following, we give a brief description of \text{Ccfomi} taken from [6]. \text{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.

\text{Ccfomi} has three kind of variables to represent the partially built \text{FO}: \text{from}_k \text{ are the set of labels of all currently existing factor links going forward from } k. \text{S}_i \text{ are suffix links from each state } i \text{ and } \delta_{\sigma_i} \text{ give the state reached from } k \text{ by following a factor link labeled } \sigma_i.

Following, we explain some \text{Ccfomi}’s processes. \text{SYNC} synchronizes the improvisation phases. In order to achieve that, it is required that the simulation phase only takes place once the \text{FO} subgraph is completely built.

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, \text{SYNC} is “waiting” until \text{go} is greater or equal than one. It means that
the \textit{PLAYER} process has played the note \(i\) and the \textit{ADD} process can add a new symbol to the \textit{FO}.

\[
\text{SYNC}_k \overset{\text{def}}{=} \begin{cases} \text{when } S_{i-1} \geq -1 \land \text{go} \geq i \text{ do} \\
(\text{ADD}_i \parallel \text{next } \text{SYNC}_{i+1}) \\
\text{unless } S_{i-1} \geq -1 \land \text{go} \geq i \text{ next } \text{SYNC}_i 
\end{cases}
\]

The \textit{PLAYER} 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 from the environment.

The improvisation process \textit{IMPROV} starts from state \(k\) and probabilistically, chooses whether to output the symbol \(O_k\) or to follow a backward link \(S_k\). We have modeled \textit{IMPROV} as a simpler improvisation process. Choices in our \textit{IMPROV} process are made non-deterministically.

\[
\text{IMPROV}(k) \overset{\text{def}}{=} \begin{cases} \text{when } S_k = -1 \text{ do next} \\
(tell(out = \sigma_k) \parallel \text{IMPROV}(k + 1)) \\
\text{when } S_k \geq 0 \text{ do next} \\
(\sum_{\sigma \in \Sigma} \text{ when } \sigma \in \text{ from}_{S_k} \text{ do} \\
\left(\text{tell(out = } \sigma) \parallel \text{IMPROV}(\delta_{S_k}, \sigma)\right) \\
\text{unless } S_k \geq -1 \text{ next } \text{IMPROV}(k) 
\end{cases}
\]

The system is modeled as the \textit{PLAYER} and the \textit{SYNC} running in parallel with a process waiting until \(n\) symbols have been played to launch the \textit{IMPROV}.

\[
\text{SYS}_n \overset{\text{def}}{=} \text{tell}(S_0 = -1) \parallel \text{PLAYER}_1 \parallel \text{SYNC}_1 \parallel \text{Wait}_n
\]

4.2. Signal Processing

\texttt{Ntcc} was used in the past as an audio processing framework [7]. 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 \texttt{ntcc}. They proposed a \texttt{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.

Our approach is different, we use a \texttt{Ntccrt} external for Pd to synchronize the graphical objects in charge of audio, video or Musical Instrument Digital Interface (MIDI) processing in Pd.

For instance, the \texttt{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 Pd objects to apply a filter on that audio stream, until the 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 (easily represented by a constraint) over its input is true.

Our system is composed by the variables \(work, end\) and \(input\), which represent lists. \texttt{Work}_j represents the identifiers of the filters working on the object \(j\). \texttt{End}_j represents when the object \(j\) has finished its work. \texttt{Input}_j represents the conditions necessary to launch filter \(P_i\). Note that the values for \texttt{end} and \texttt{input} are updated each time-unit with information from the environment. Finally, \texttt{wait}_j represents the set of filters waiting to work on the object \(m_j\).

Next, we explain the definitions of our system. Objects are represented by \texttt{IdleObject} and \texttt{BusyObject}. An object is \texttt{idle} until it non-deterministically chooses a filter from the \texttt{wait}_j variable. After that, it will remain \texttt{busy} until the \texttt{end}_j = \texttt{true} constraint can be deduced from the store.

\[
\texttt{IdleObject}(j) \overset{\text{def}}{=} \begin{cases} \text{when } work_j > 0 \text{ do next } \texttt{BusyObject}(j) \\
\text{unless } work_j > 0 \text{ next } \texttt{IdleObject}(j) \\
\end{cases}
\]

\[
\texttt{BusyObject}(j) \overset{\text{def}}{=} \begin{cases} \text{when } end_j = \texttt{true} \text{ do } \texttt{IdleObject}(j) \\
\text{unless } end_j = \texttt{true} \text{ next } \texttt{BusyObject}(j) 
\end{cases}
\]

Filters are represented by \texttt{IdleFltr}, \texttt{WaitingFltr} and \texttt{BusyFltr} (fig. 1). A filter is \texttt{idle} until it can deduce that \texttt{input}_i = \texttt{true}. Analogously, a filter is \texttt{busy} until it can deduce that the filter finished working on the object associated to it. \texttt{Input}_j and \texttt{end}_j can be a conditions based on multiple criteria.

\[
\texttt{IdleFltr}(i, j) \overset{\text{def}}{=} \begin{cases} \text{when } \texttt{input}_i = \texttt{true} \text{ do } \text{WaitingFltr}(i, j) \\
\text{unless } \texttt{input}_i = \texttt{true} \text{ next } \text{IdleFltr}(i, j) 
\end{cases}
\]

A filter is \texttt{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 \texttt{BusyFltr}.

\[
\texttt{WaitingFltr}(i, j) \overset{\text{def}}{=} \begin{cases} \text{when } work_j = i \text{ do } \text{BusyFltr}(i, j) \\
\text{unless } work_j = i \text{ next } \text{WaitingFltr}(i, j) \parallel \text{tell } i \in \text{wait}_j 
\end{cases}
\]

The following definition models a system with two objects and four filters working concurrently.

\[
\texttt{System}() \overset{\text{def}}{=} \begin{cases} \texttt{IdleObject}(1) \parallel \texttt{IdleObject}(2) \parallel \texttt{IdleFltr}(1, 1) \\
\text{IdleFltr}(2, 1) \parallel \texttt{IdleFltr}(2, 1) \parallel \texttt{IdleFltr}(2, 2) 
\end{cases}
\]
5. 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 [4] that an improvisation system able to learn and produce sequences in less than 30 ms is appropriate for real-time interaction. Therefore, running Ccfomi, Ntccrt is capable of real-time interaction for a simulation of at most 300 time-units.

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.

6. 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 and a signal processing system. We ran both applications using 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-friendly 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 paradigm of Pd.

We also argue that using the graphical paradigm is difficult and is 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 a few lines in Common Lisp, representing complex conditions with constraints.

We encourage formal verification for ntcc, but there is not a tool to verify ntcc models. In the future, we propose developing verification tools for ntcc models.

7. 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.

8. REFERENCES