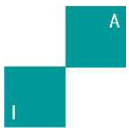
Inteligencia Artificial 19(57) (2016), 37-73
doi: 10.4114/ia.v18i56.1122



INTELIGENCIA ARTIFICIAL

http://journal.iberamia.org/

Probabilistic Extension to the Concurrent Constraint Factor Oracle Model for Music Improvisation

Mauricio Toro
Universidad Eafit
Carrera 49 # 7 sur - 50, Medellín, Antioquia, Colombia
mtorobe@eafit.edu.co

Abstract

We can program a Real-Time (RT) music improvisation system in C++ without a formal semantic or we can model it with process calculi such as the Non-deterministic Timed Concurrent Constraint (ntcc) calculus. "A Concurrent Constraints Factor Oracle (FO) model for Music Improvisation" (Ccfomi) is an improvisation model specified on ntcc. Since Ccfomi improvises non-deterministically, there is no control on choices and therefore little control over the sequence variation during the improvisation. To avoid this, we extended Ccfomi using the Probabilistic Non-deterministic Timed Concurrent Constraint calculus. Our extension to Ccfomi does not change the time and space complexity of building the FO, thus making our extension compatible with RT. However, there was not a ntcc interpreter capable of RT to execute Ccfomi. We developed Ntccrt –a RT capable interpreter for ntcc– and we executed Ccfomi on Ntccrt. In the future, we plan to extend Ntccrt to execute our extension to Ccfomi.

Resumen

Podemos modelar un sistema de improvisación musical de tiempo real (RT) en C++ sin una semántica formal o podemos modelarlo con un cálculo de procesos como el cálculo de procesos temporal, no determinístico, con restricciones (ntcc). "Un modelo de improvisación musical concurrente por restricciones basado en el Oráculo de Factores (FO)" (Ccfomi) es un modelo de improvisación especificado en ntcc. Como Ccfomi improvisa de una manera no determinística, no hay control sobre las escogencias que hace y por consiguiente, hay poco control sobre la variación en las secuancias durante la improvisación. Para evitar esto, proponemos extender Ccfomi usando el cálculo de procesos temporal, no determinístico, probabilístico con restricciones. Nuestra extensión a Ccfomi no cambia la complejidad en tiempo y espacio del algoritmo de construcción del FO, haciendo nuestra extensión compatible con RT. Sin embargo, no hay un intérprete para ntcc que sea capáz de ejecutar Ccfomi en RT. Nosotros desarrollamos Ntccrt –nuestro intérprete para ntcc capaz de RT– y ejecutamos Ccfomi en Ntccrt. En el futuro, planeamos extender Ntccrt para ejecutar nuestra extensión a Ccfomi.

Keywords: Factor oracle, concurrent constraints programming, ccp, machine learning, machine improvisation, Ccfomi, Gecode, ntcc, pntcc, real-time.
Palabras Clave: Oráclulo de factores, programación concurrente por restricciones, ccp, aprendizaje por computador, improvisación musical por computador, Ccfomi, Gecode, ntcc, pntcc, tiempo real.

1 Introduction

There are two different approaches to develop multimedia interaction systems (e.g., machine improvisation).

ISSN: 1988-3064(on-line)

One may think that in order to implement real-time capable systems, those systems should be written directly in C++ for efficiency. In contrast, one may argue that multimedia interaction systems –inherently concurrent– should not be written directly in C or C++ because there is not a formalism to reason about concurrency in C++. We argue that those systems should be modeled using a process calculus with formal semantics and verification procedures, and execute those models on a real-time capable interpreter. That will be our definition for real-time in the rest of this document.

Garavel explains in [9] that models based on process calculi are not widespread because there are many calculi and many variants for each calculus, being difficult to choose the most appropriate. In addition, it is difficult to express an explicit notion of time and real-time requirements in process calculi. Finally, he argues that existing tools for process calculi are not user-friendly.

1.1 Motivation

Defending the calculi approach, Rueda et al. [27],[28] explain that using the semantics and logic underlying the Non-deterministic Timed Concurrent Constraint (`ntcc`) [17] calculus, it is possible to prove properties of the `ntcc` models before executing them and execute the models on a `ntcc` interpreter. We define soft real-time multimedia interaction means that the system reacts fast enough to interact with human players without letting them notice delays.

One may disagree with Rueda et al., arguing that although there are several interpreters for `ntcc` such as Lman [16] and Rueda's Interpreter [28], there is not a generic interpreter to run `ntcc` models in real-time.

We agree with Rueda et al. about the way to develop those systems, but we also argue that currently there are no `ntcc` interpreters capable of real-time. We argue, in agreement with Rueda et al.'s argument, that models based on `ntcc` such as *"A Concurrent Constraints Factor Oracle model for Music Improvisation"* (*Ccfomi*) [28] are a good alternative to model multimedia interaction because synchronization is presented declaratively by means of variable sharing among concurrent agents reasoning about information contained in a global *store*. However, due to non-deterministic choices, improvisation in Ccfomi can be repetitive (i.e., it produces loops without control). In addition, since Ccfomi does not change the intensity of the learned notes, Ccfomi may produce a sharp difference in the relative loudness between what a musician plays and what the improviser plays.

Process calculi has been applied to the modeling of interactive music systems. As an example, process calculi was proved succesful to model interactive scores [37] and Temporal Relations for Micro and Macro Controls [36]. Process calculi has also been used to model ecological systems, for instance, population models for dengue [38] and ecological systems [21].

Our main objective is extending *Ccfomi* to model probabilistic choice of musical sequences. We also want to show that a `ntcc` model can interact with a human player in soft real-time using a `ntcc` interpreter. For that reason, we developed Ntccrt, a generic real-time interpreter for `ntcc`.

The rest of this introduction is organized as follows. First section, gives a definition of music improvisation. Second section, explains machine improvisation. Third section, gives a brief introduction to `ntcc` and presents systems modeled with `ntcc`. After explaining the intuitions about music improvisation, machine improvisation, and `ntcc` we explain our solution to extend *Ccfomi* in Section fourth section. Fifth section explains the contributions of this article. Finally, sixth section explains the organization of the following chapters.

1.2 Music improvisation

"Musical improvisation is the spontaneous creative process of making music while it is being performed. To use a linguistic analogy, improvisation is like speaking or having a conversation as opposed to reciting a written text. Among jazz musicians there is an adage, *improvisation is composition speeded up*, and vice versa, *composition is improvisation slowed down*."[14]

Improvisation exists in almost all music generel. However, improvisation is most frequently associated with melodic improvisation as it is found in jazz. However, spontaneous real-time variation in performance of tempo and dynamics within a classical performance may also be considered as improvisation [14].

1.3 Machine improvisation

Machine improvisation is the simulation of music improvisation by the computer. This process builds a representation of music, either by explicit coding of rules or applying machine learning methods. For real-time machine improvisation it is necessary to perform two phases concurrently: *Stylistic learning* and *Stylistic simulation*. In addition, to perform both phases concurrently, the system must be able to interact in real-time with human players [28].

Rueda et al. define *Stylistic learning* as the process of applying such methods to musical sequences in order to capture important musical features and organize these features into a model, and the *Stylistic simulation* as the process producing musical sequences stylistically consistent with the learned style [28]. An example of a system running concurrently both phases is *Ccfomi*, a system using the Factor Oracle (FO) to store the information of the learned sequences and the `ntcc` calculus to synchronize both phases of the improvisation.

The `ntcc` calculus is a mathematic formalism used to represent reactive systems with synchronous, asynchronous and/or non-deterministic behavior. This formalism and its extensions have been used to model systems such as: musical improvisation systems [28] and an audio processing framework [29].

1.4 Our solution

To avoid a repetitive improvisation, we extend *Ccfomi* with the Probabilistic Non-deterministic Timed Concurrent Constraint (`pntcc`) calculus [20] to decrease the probability of choosing a sequence previously improvised. This idea is based on the Probabilistic *Ccfomi* model [20] developed by Pérez and Rueda. That model, chooses the improvised sequences probabilistically, based on a probability distribution. Unfortunately, Probabilistic *Ccfomi* does not give information about how that probability distribution can be built nor how it can change through time according to the user and the computer interaction. Our model is the first `pntcc` model, as far as we know, where probability distributions change from a time-unit to another.

For instance, consider that our system can play in a certain moment the pitches (i.e., the frecuency of the notes) a,b and c with an equal probability. Then it outputs the sequence "*aaba*". After that, it is going to choose another pitch. When choosing this pitch, c has a greater probability to be chosen than b, and b has a greater probability to be chosen than a because a was played three times and b once in the last sequence. Using this probabilistic extension, we avoid multiple cycles in the improvisation which can happen without control in *Ccfomi*.

On the other hand, to be coherent with the relative loudness on which the user is currently playing, we change the intensity of the improvised notes. This idea is based on interviews with musicians Riascos and Juan Manuel Collazos, where they argue that this is a technique they use when improvising and improves the "quality" of the improvisation, when two or more persons improvise at the same time.

For instance, if the computer plays five notes with intensities (measured from 0 to 127) 54, 65, 30, 58, 91 and the user plays, at the same time, four notes with intensities 10, 21, 32, 5; they are incoherence results because the user is playing low and the computer is playing loud. For that reason, our system multiplies its intensities by a factor of 0.29 (the relation of the average of both sequences) changing the intensities of the computer output to 16, 19, 9, 17, 26.

1.5 Contributions

- **Ntccrt**. A real-time capable interpreter for `ntcc`[35]. Using Ntccrt, we executed *Ccfomi*. Examples, sources and binaries can be found at http://ntccrt.sourceforge.net. An article about Ntccrt is to be published this year.

- **Gelisp**. A new graphical constraint solving library for OpenMusic. We plan to use it in the future for a closer integration between *Ntccrt* and OpenMusic. Examples, sources, and binaries can be found at http://gelisp.sourceforge.net. An article about Gelisp is to be publish this year. The original version of Gelisp was developed by Rueda for Common Lisp [39].

1.6 Organization

The structure of this article is the following. In Chapter 1, we explain the background concepts. Chapter 2 focuses on the modeling of *Ccfomi* to allow probabilistic choice of musical sequences. Chapter 3 explains the modifications to *Ccfomi* to allow variation of the intensity of learned notes during the style simulation phase. Chapter 4 describes our model in `pntcc`. Chapter 5 explains the design and implementation of Ntccrt, our real-time interpreter for `ntcc`. Chapter 6 shows some results and tests made with the interpreter. Finally, in Chapter 7, we present a summary of this article, concluding remarks, and propose some future work.

2 Background

2.1 Concurrent Constraint Programming (CCP)

Concurrent Constraint Programming (CCP [31]) is a model for concurrent systems. In CCP, a concurrent system is modeled in terms of constraints over the system variables and in terms of agents interacting with partial information obtained from those variables. A constraint is a formula representing partial information about the values of some of the system variables. Programming languages based on the CCP model, provide a *propagator* for each user-defined constraint.

Propagators can be seen as operators reducing the set of possible values for some variables. For instance, in a system with variables $pitch_1$ and $pitch_2$ taking Musical Instrument Digital Interface (MIDI) values, the constraint $pitch_1 > pitch_2 + 2$ specifies possible values for $pitch_1$ and $pitch_2$ (where $pitch_1$ is at least one tone higher than $pitch_2$). In MIDI notation, each MIDI pitch unit represents a semi-tone.

The CCP model includes a set of constraints and a relation to know when a constraint can be deduced from others (named entailment relation $\models$). This relation gives a way of deducing a constraint from the information supplied by other constraints.

"The idea of the CCP model is to accumulate information in a *store*. The information on the *store* can increase but it cannot decrease. Concurrent processes interact with the *store* by either adding more information or by asking if some constraint can be deduced from the current *store*. If the constraint cannot be deduced, this process blocks until there is enough information to deduce the constraint" [28].

Consider for instance four agents interacting concurrently (fig. 1). The processes **tell** ($pitch_1 > pitch_2 + 2$) and **tell**($pitch_2 > 60$) add new information to the *store*. The processes **ask**($pitch_1 > 58$) **do** P and **ask**($pitch_1 = 58$) **do** Q launch process P and Q (P and Q can be any process) respectively, when their condition can be entailed from the *store*. After the execution of the **tell** processes, process **ask** ($pitch_1 > 58$) **do** P launches process P, but the process **ask** ($pitch_1 = 58$) **do** Q will be suspended until its condition can be entailed from the *store*.

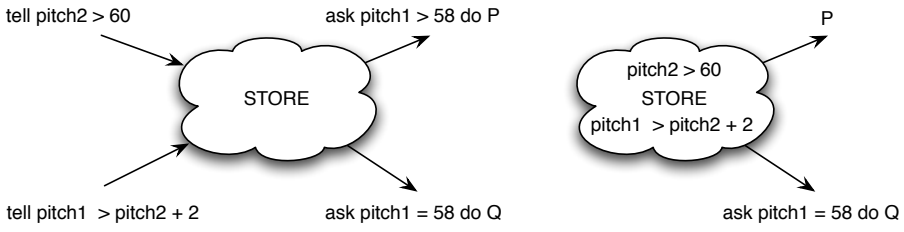


Figure 1: Process interaction in CCP

Formally, the CCP model is based on the idea of a constraint system. "A constraint system is a structure $< D, \vdash, Var >$ where D is a (countable) set of primitive constraints (or tokens), $\vdash \in D \times D$ is an inference relation (logical entailment) that relates tokens to tokens and Var is an infinite set of variables" [31]. A (non primitive) constraint can be composed out of primitive constraints.

The formal definition of CCP does not specify which types of constraints can be used. Thus, a constraint system can be adapted to a particular need depending on the set D. For instance, *finite*

Agent	Meaning
tell (c)	Adds the constraint 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	Non deterministically chooses P_i s.t. (c_i) holds
$*P$	Delays P indefinitely (not forever)
$!P$	Executes P each *time-unit* (from now)

Table 1: `Ntcc` Agents

domain (FD) constraint system provides primitive constraints (also called *basic constraints*) such as $x \in R$, where R is a set of ranges of integers. On the other hand, *finite set (FS)* constraint system provides primitive constraints such as $y \in S$, where S is a set of FD variables and y is an FD variable. Constraints systems may also include expressions over trees, graphs, and sets.

Valencia and Rueda argue in [30] that the CCP model posses difficulties for modeling reactive systems where information on a given variable changes depending on the interactions of a system with its environment. The problem arises because information can only be added to the *store*, not deleted nor changed.

2.2 Non-deterministic Timed Concurrent Constraint (`ntcc`)

`Ntcc` introduces 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. In `ntcc` we can have a variable x taking different values along *time-units*. To model that in CCP, we would have to create a new variable x_i each time we change the value of x.

Following, we give some examples of how the computational agents of `ntcc` can be used. The operational semantic of all `ntcc` agents can be found in Appendix **??** and a summary can be found in table 1. Using the **tell** agent with a FD constraint system, it is possible to add constraints such as **tell**$(pitch_1 = 60)$ (meaning the $pitch_1$ must be equal to 60) or **tell**$(60 < pitch_2 < 100)$ (meaning that $pitch_2$ is an integer between 60 and 100).

The **when** agent can be used to describe how the system reacts to different events, for instance **when** $pitch_1 = 48 \wedge pitch_2 = 52 \wedge pitch_3 = 55$ **do tell**$(CMayor = true)$ is a process reacting as soon as the pitch sequence C, E, G (represented as 48, 52, 55 in MIDI notation) 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** $(pitch_1 = 52) \parallel$ **when** $48 < pitch_1 < 59$ **do tell** $(Instrument = 1)$ is a process telling the *store* that $pitch_1$ is 62 and concurrently reacts when $pitch_1$ is in the octave -1, assigning *instrument* to 1 (fig. 2). The number one represents the acoustic piano in MIDI notation.

The **next** agent 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** agent 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* (i.e., $pitch_1$ was not played in the current *time-unit*), telling the *store* in the next *time-unit* that *lastpitch* is not 60 (fig. 3).

The ***** agent may be used in music to delay the end of a music process indefinitely, but not forever (i.e., we know that the process will be executed, but we do not know when). For instance, $*$**tell** $(End = true)$. The ! agent executes a certain process in every *time-unit* after its execution. For instance, !**tell** $(PlaySong = true)$. The $\sum$ agent is used to model non-deterministic choices. For instance, ! $\sum_{i \in \{48,52,55\}}$

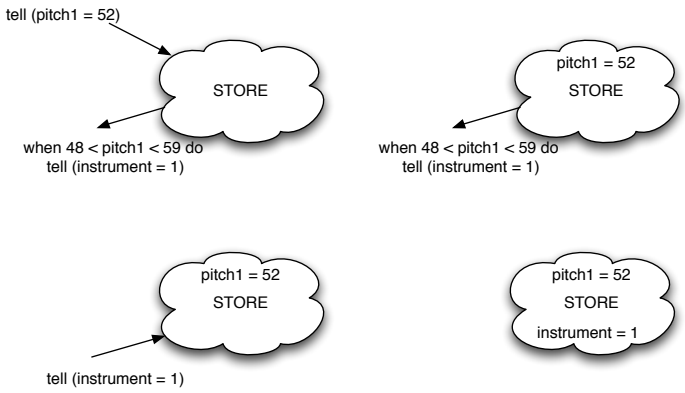


Figure 2: **Tell**, **when**, and **parallel** agents in *Ntcc*

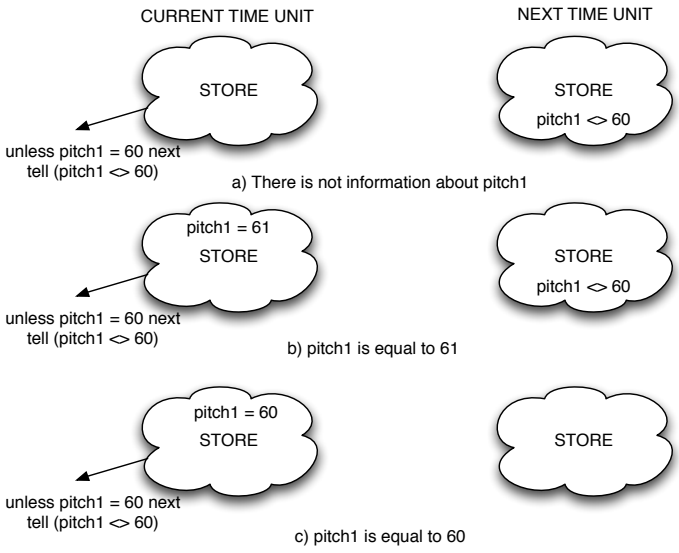


Figure 3: Unless agent in `ntcc`

when *true* **do tell** $(pitch = i)$ models a system where each *time-unit*, a note is chosen from the C major chord (represented by the MIDI numbers 48,52 and 55) to be played (fig. 4[1]).

The agents presented in table 2 are derived from the basic operators. The agent $A + B$ non-deterministically chooses to execute either A or B. The **persistent assignation** process $x \leftarrow t$ changes the value of x to the current value of t in the following *time-units*. In a similar way, the agents in table 3 are used to model cells. Cells are variables which value can be re-assigned in terms of its previous value. $x : (z)$ creates a new cell x with initial value z, $x :\leftarrow g(x)$ changes the value of a cell (this is different from $x \leftarrow t$ which changes the value of x only once), and $exch_g[x, y]$ exchanges the value of cell x and z.

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 we do not want an infinite 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. Further information about recursion in `ntcc` can

[1] $!\sum_{i\in\{48,52,55\}}$ **when** *true* **do tell** $(pitch = i)$ can be expressed as !(**tell** $(pitch = 48)$ + **tell** $(pitch = 52)$ + **tell** $(pitch = 55)$)

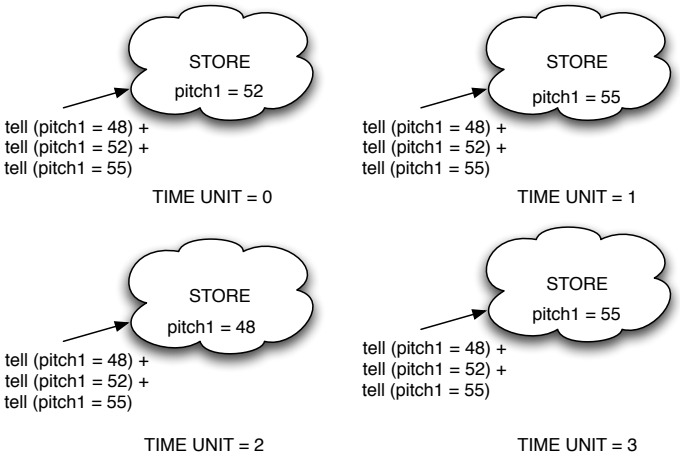


Figure 4: Execution of a non-deterministic process in `ntcc`

Agent	Meaning
A + B	$\sum_{i \in 1,2}$ **when** *true* **do** (**when** $i = 1$ **do** $A \parallel$ **when** $i = 2$ **do** B)
$x \leftarrow t$	**local** v **in** $\sum_{v}$ **when** $t = v$ **do next !tell** $(x = v)$

Table 2: Derived `ntcc` agents

be found at [17].

2.3 Generic Constraint Development Environment (Gecode)

Gecode is a constraint solving library written in C++. Gecode is based on Constraints as Propagation agents (CPA) according to [39]. A CPA system provides multiple propagators to transform a (non-primitive) constraint into primitive constraints supplying the same information. In a finite domain constraint system, primitive constraints have the form $x \in [a..b]$. For instance, in a *store* containing $pitch_1 \in [36..72]$, $pitch_2 \in [60..80]$, a propagator $pitch_1 > pitch_2 + 2$ would add constraints $pitch_1 \in [63..72]$ and $pitch_2 \in [60..69]$.

The reader may notice that there is a similarity between CPA and `ntcc`. Both of them are based on concurrent agents working over a constraint *store*. In chapter 6, we explain how we can encode `ntcc` agents as *propagators*.

Gecode works on different operating systems and is currently being used as the constraint library for Alice[25] and soon it will be used in Mozart-Oz, therefore it will be maintained for a long time. Furthermore, it provides an extensible API, allowing us to create new *propagators*. Finally, we conjecture

Agent	Meaning
$x : (z)$	**tell**$(x = z) \parallel$ **unless** change(x) **next** $x : (z)$
$x :\leftarrow g(x)$	**local** $v \sum_{v}$ **when** $x = v$ **do** (**tell** change$(x) \parallel$ **next** $x : g(v)$)
$exch_g[x, y]$	**local** $v \sum_{v}$ **when** $t = v$ **do** (**tell**(change$(x) \parallel$ (**tell**(change(y) $\parallel$ **next** $(x : g(v) \parallel y : (v))$

Table 3: Definition of cells

that *Gecode*'s performance is better than the constraints solving tool-kits used in Sicstus Prolog and Mozart-Oz based on Gecode's benchmarks[2].

2.4 Factor Oracle (*FO*)

The Factor Oracle (*FO*)[1] is a finite automaton that can be built in linear time and space, in an incremental fashion. The *FO* recognizes at least all the sub-sequences (factors) of a given a sequence (it recognizes other sequences that are not factors). All the states of the *FO* are considered as accepting states. A sequence of symbols s = $\sigma_1\sigma_2...\sigma_n$ is learned by such automaton, which states are 0,1,2...n. There is always a transition arrow (called *factor link*) from the state $i-1$ to the state i and there are some transition arrows directed "backwards", going from state i to j (where $i > j$), called *suffix links*. *Suffix links*, opposed to *factor links*, are not labeled. For instance, a *FO* automaton for s = ab is presented in Figure 5, where black headed arrows represent the *factor links* and white headed arrows represent the *suffix links*.

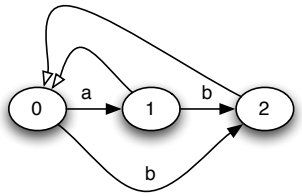


Figure 5: A *FO* automaton for $s = ab$

The *FO* is built on-line and their authors proved that its algorithm has a linear complexity in time and space[1]. For each new entering symbol σ, a new state i is added and an arrow from $i-1$ to i is created with label σ_i. Starting from $i-1$, the *suffix links* are iteratively followed backward, until a state is reached where a *factor link* with label σ_i going to some state j, or until there are no more *suffix links* to follow. For each state met during this iteration, a new *factor link* labeled by σ_i is added from this state to i. Finally, a *suffix link* is added from i to state j or to state 0 depending on which condition terminated the iteration. Further formal definitions and the proof of *FO* complexity can be found in [1]. The on-line construction algorithm is presented with detail in Appendix **??**

Since the *FO* has a linear complexity in time and space, it was found in [10] that it is appropriate for machine improvisation. In addition, all attribute values for a music event can be kept in an object attached to the corresponding node, since the actual information structure is given by the configuration of arrows (*factor and suffix links*). Therefore a tuple with *pitch* (the frecuency of the note), *duration* (the amount of time that the note is played), and *intensity* (the volume on which is the note is played) can be related to each arrow according to [10].

2.5 Concurrent Constraint Factor Oracle Model for Music Improvisation

Concurrent Constraint Factor Oracle Model for Music Improvisation ($Ccfomi$) is defined in [28]. Following, we present a briefly explanation of the model taken from [28]. $Ccfomi$ has three kinds 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 the *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. For instance, the *FO* in figure 5 is represented by $from_0 = \{a, b\}$,$from_1 = \{b\}$, $S_1 = 0$, $S_2 = 0$, $\delta_{0,a} = 1$, $\delta_{0,b} = 2$.

Although it is not stated explicitly in $Ccfomi$, the variables $from_k$ and δ_{k,σ_i} are modeled as infinite rational trees [24] with unary branching, allowing us to add elements to them, each *time-unit*. Infinite rational trees have infinite size. However, they only contain a finite number of distinct sub-trees. For that reason, they have been subjects of multiple axiomatizations to construct a constraint system based

[2]Benchmarks presented in http://www.gecode.org

on them. For instance, posting the constraints $cons(c, nil, B)$, $cons(b, B, C)$, $cons(a, C, D)$ we can model a list of three elements $[a, b, c]$.

Ccfomi is divided in three subsystems: learning (ADD), improvisation (CHOICE) and playing (PLAYER) running concurrently. In addition, there is a synchronization process (SYNC) that takes care of synchronization.

The ADD process is in charge of building the FO (this process models the learning phase) by creating the *factor links* and *suffix links*. Note that the process ADD calls the $LOOP$ process.

$$ADD_i \stackrel{def}{=} \textbf{!tell}(\delta_{i-1,\sigma_i} = i) \parallel LOOP_i(S_{i-1})$$

"Process $LOOP_i(k)$ adds (if needed) *factor links* labeled σ_i to state i from all states k reached from $i-1$ by *suffix links*, then computes S_i, the *suffix link* from i" [28].

$$\begin{array}{l} LOOP_i(k) \stackrel{def}{=} \\ \quad \textbf{when } k \geq 0 \textbf{ do} \\ \quad\quad \textbf{unless } \sigma_i \in from_k \\ \quad\quad\quad \textbf{next}(\textbf{!tell } (\sigma_i \in from_k) \parallel \textbf{!tell}(\delta_{k,\sigma_i} = i) \parallel LOOP_i(S_k)) \\ \quad \parallel \textbf{when } k = -1 \textbf{ do } \textbf{!tell}(S_i = 0) \\ \quad \parallel \textbf{when } k \geq 0 \wedge \sigma_i \in from_k \textbf{ do } \textbf{!tell}(S_i = \delta_{k,\sigma_i}) \end{array}$$

"A musician is modeled as a $PLAYER$ process playing some note p every once in a while. The $PLAYER$ process non-deterministically chooses between playing a note now or postponing the decision to the next *time-unit*" [28].

$$\begin{array}{l} PLAYER_j \stackrel{def}{=} \\ \quad \sum_{p \in \Sigma} \textbf{when } \text{true } \textbf{do } (\textbf{!tell}(\sigma_j = p) \parallel \textbf{tell}(go = j) \parallel \textbf{next } PLAYER_{j+1}) \\ \quad + (\textbf{tell } (go = j - 1) \parallel \textbf{next } PLAYER_j) \end{array}$$

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 sub-graph is completely built. The $SYNC_i$ 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 used of constraints. When a variable has no value, the *when* processes depending on it are blocked. Therefore, the $SYNC_i$ process is "waiting" until go is greater or equal than one. It means that the $PLAYER_i$ process has played the note i and the ADD_i process can add a new symbol to the FO. The other condition $S_{i-1} \geq -1$ is because the first *suffix link* of the FO is equal to -1 and it cannot be followed in the simulation phase.

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

"The improvisation process $CHOICE_\Phi(k)$ uses the distribution function $\Phi : \mathbb{R} \to \{0, 1\}$. The process starts from state k and stochastically, chooses according to probability q, whether to output the symbol σ_k or to follow a backward link S_k"[28].

$$\begin{array}{l} CHOICE_\Phi(k) \stackrel{def}{=} \\ \quad \textbf{when } S_k = -1 \textbf{ do next}(\textbf{ tell } (out = \sigma_{k+1}) \parallel CHOICE_\Phi(k+1)) \\ \quad \parallel \textbf{tell } (flip = \Phi_k(q)) \\ \quad \parallel \textbf{when } flip = 1 \wedge S_{k+1} \geq 0 \textbf{ do next } (\textbf{tell } (out = \sigma_{k+1}) \parallel CHOICE_\Phi(k+1)) \\ \quad \parallel \textbf{unless } flip = 1 \wedge S_{k+1} \geq 0 \end{array}$$

$$\textbf{next} \sum_{\sigma \in \Sigma} \textbf{when } \sigma \in from_{s_k} \textbf{ do } (\textbf{ tell } (out = \sigma) \parallel CHOICE_{\Phi}(\delta_{s_k,\sigma})$$

The whole system is represented by a process doing all the initializations and launching the processes when corresponding. Improvisation starts after n symbols have been created by the $PLAYER$ process.

$System_{n,p} \stackrel{def}{=}$
$\quad$ **!tell**$(q = p) \parallel$ **!tell**$(S_0 = -1) \parallel PLAYER_1 \parallel SYNC_1$
$\quad \parallel$ **!when** $go = n$ **do** $CHOICE(n)$

2.6 Probabilistic Non-deterministic Timed Concurrent Constraint (pntcc)

"One possible critique to CCP is that it is too generic for representing certain complex systems. Even if counting with partial information is extremely valuable, we find that properly taking into account certain phenomena remains to be difficult, which severely affects both modeling and verification. Particularly challenging is the case of uncertain behavior. Indeed, the uncertainty underlying concurrent interactions in areas such as computer music goes way beyond of what can be modeled using partial information only." [20].

The first attempt to extend `ntcc` to work with probabilities was the *Stochastic Non-deterministic Timed Concurrent Constraint (`sntcc` [18])* calculus. `Sntcc` provides an operator P_ρ to decide whether to execute or not a process with a certain probability ρ. Using `sntcc`, *Ccfomi* models the action of choosing between a *suffix link* or a *factor link* with a probability ρ. However, when using `sntcc`, it is not possible to use a probability distribution to choose among all the *factor links* following a state in the *FO*. The probability distribution describes the range of possible values that a random variable can take.

`Pntcc` overcomes that problem, it provides a new agent to the calculus for probabilistic choice $\bigoplus$. The probabilistic choice $\bigoplus$ operator has the following syntax:

$$\bigoplus_{i \in I} \textbf{when } C_i \textbf{ do } (P_i, a_i),$$

where I is a finite set of indexes, and for every $a_i \in \mathbb{R}^{(0.0,1.0]}$ we have $\sum_{i \in I} a_i = 1.0$.

"The intuition of this operator is as follows. Each a_i associated to P_i represents its probability of being selected for execution. Hence, the collection of all a_i represents a probability distribution. The guards that can be entailed from the current *store* determine a subset of enabled processes, which are used to determine an eventual normalization of the a_i's. In the current time interval, the summation probabilistically chooses one of the enabled process according to the distribution defined by the (possibly normalized) a_i's. The chosen alternative, if any, precludes the others. If no choice is possible then the summation is precluded." [20].

Using the probabilistic choice we can model a process choosing a *factor link* from the FO with a probability distribution ρ.

$$\bigoplus_{\sigma \in \sum} \textbf{when } \sigma \in from_k \textbf{ do } (\textbf{tell}(output = \sigma), \rho_\sigma)$$

The operational semantic of the $\bigoplus$ agent and other formal definitions about `pntcc` can be found in Appendix **??**.

3 Probabilistic Choice of Musical Sequences

When modeling machine improvisation, we want to choose a certain music sequence, based on the history of user and computer interaction. For instance, when traversing the Factor Oracle (FO) in the simulation phase, we want some information to choose among the *factor links* and the *suffix link* following a certain state. To achieve that, we propose to assign integers to the links in the *FO*. Using those integers, we can calculate probabilities to choose a link based on a probability distribution. We recall from the introduction that our main objective is extending *Ccfomi* to model probabilistic choice of musical sequences.

In the beginning of this article, we developed a probabilistic model which changes the complexity in time for building the *FO* to quadratic (see Appendix **??**). The idea behind it was changing the probabilities of all the *factor links* coming from state i when modifying a *factor link* leaving from that state. This idea was discarded for not being compatible with soft real-time (consider soft real-time as defined in the introduction).

The probabilistic model we chose is based on a simple, yet powerful concept. Using the system parameters, the probability of choosing a *factor link* in the simulation phase will decrease each time a *factor link* is chosen. Additionally, we calculate the length of the common suffix (*context*) associated to each *suffix link*. Using the *context*, we reward the *suffix links*. Further information about the *context* can be found at [13].

We represent the system with four kind of variables used to represent: the *FO* states and transitions; the musical information attached to the *FO*; the probabilistic information; and the information to change musical attributes in the notes, based on the user style.

In addition to the variables described before, the system has some information parametrized by the user: $\alpha, \beta, \gamma, \tau$ and n. The constant α is the recombination factor, representing the proportion of new sequences desired. β represents the factor for decreasing the importance of a *factor link* when it is chosen in the simulation phase. γ represents the importance of a new *factor link* in relation with the other *factor links* coming from the same state. τ (described in Chapter 4) is a parameter for changing musical attributes in the notes. Finally, n is a parameter representing the number of notes that must be learned before starting the simulation phase. In Chapter 5 we describe how can we use n to synchronize the improvisation phases.

We label each *factor link* by the *pitch*. Moreover, outside the *FO* definition, we create a tuple of three integers for each *factor link*: *pitch*, *duration*, and *intensity*. These three characteristics are represented by integers. The *pitch* and the *intensity* are represented by integers from 0 to 127 and the *duration* is represented by milliseconds[3]. That way we can build a *pitch FO* (i.e., a *FO* where the symbols are pitches) associating to it other musical information.

At the same time we build a *FO*, we also create three integer arrays: ρ, $\mathbb{C}$ and *sum*. There is an integer $\rho_{i,\sigma}$ for every *factor link*, $\mathbb{C}_i$ for every *suffix link*, and sum_i for every state i. Note that $\rho_{i,\sigma}/sum_i$ would represent the probability of choosing a *factor link* if *suffix links* were not considered, and $\mathbb{C}_i$ is the *context*.

Next, we show the learning and simulation phases for the probabilistic extension. We present some simple examples explaining how the probabilities are calculated in the learning phase and how they are used in the simulation phase. Finally, we present some concluding remarks and other improvisation models related to our model.

3.1 Stylistic learning phase

During the learning phase we store an integer $\rho_{i,\sigma}$ for each *factor link* going from i labelled by σ. We also store an integer sum_i for each state i of the automaton.

The initial value for $\rho_{i,\sigma}$ is $sum_i * \gamma$ (fig. 6), where γ is a system parameter representing the importance of a new sequence in relation with the sequences already learned. When a *factor link* from i labeled by σ_j is the first *factor link* leaving from i, we assign to sum_i and ρ_{i,σ_j} the constant c. We want c to be a big integer, allowing us to have more precision when reasoning about $\rho_{i,\sigma_j}/sum_i$.

The reader may notice that this approach gives a certain importance to a new *factor link* leaving from i labeled by σ_j, without changing the value of all the other quantities $\rho_{i,\sigma}$ leaving from i. Furthermore, we preserve the sum of all the values $\rho_{i,\sigma}$ in the variable sum_i, for each state i. This system exhibits a very important property: For each state i, $\sum_{\sigma \in from_i} [\rho_{i,\sigma}/sum_i] = 1$. The sum of all the probabilities associated to the *factor links* coming from the same state are equal to one. This property is preserved, when changing the values of $\rho_{i,\sigma}$ and sum_i in both improvisation phases.

On the other hand, we give rewards to the *suffix link* using the *context*. To calculate the *context*, Lefebvre and Lecroq modified the *FO* construction algorithm, conserving its linear complexity in time and

[3]Pitch, duration and intensity are represented according to MIDI 1.0 standard

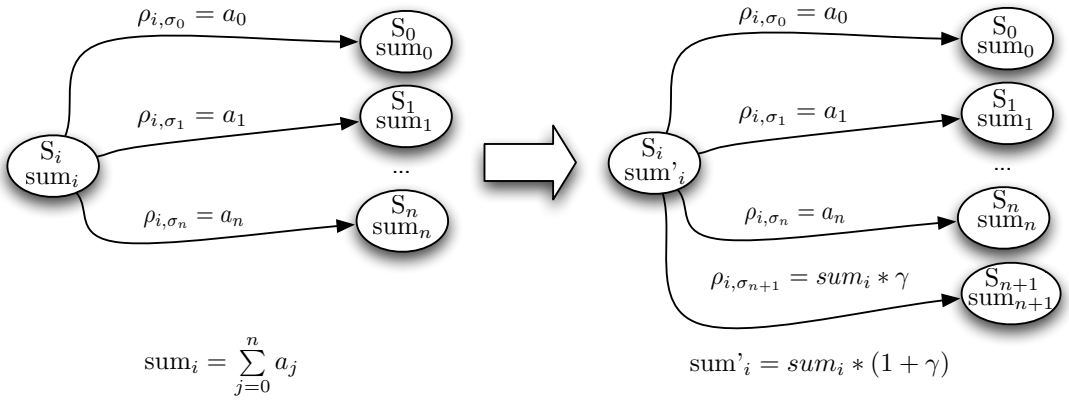


Figure 6: Adding a *factor link* to the *FO*

space [13]. This approach has been successfully used by Cont, Assayag and Dubnov on their anticipatory improvisation model [7].

Figure 7 is a simple example of a *FO* and the integer arrays presented previously. First, we present the score of a fragment of the Happy Birthday song; then we present a sequence of possible tuples $<$ *pitch*, *duration*, *intensity* $>$ for that fragment; and finally the *FO* with the probabilistic information.

3.2 Stylistic simulation phase

In the simulation phase, we use all the information calculated in the learning phase to choose the notes probabilistically. *Factor links* chosen in this phase, will decrease the importance proportionally to β. In addition, the probability of choosing secondary *factor links* is proportional to γ . We consider primary *factor links* those going from the state i to $i + 1$, and all the others as secondary . On the other hand, the *suffix links* are rewarded by the *context*, calculated on-line in the learning phase.

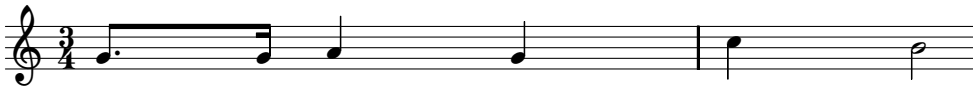
If there were not *suffix links*, we would choose a *factor link* leaving from the state i with a probability distribution $\varphi(i, \sigma)$ such that $\varphi(i, \sigma) = \rho_{i,\sigma}/sum_i$. Later on, we will explain how we can extend this concept to work with the *suffix links*, rewarded by their *context*. However, the concept of decreasing the probability of a *factor link* when it is chosen remains invariant.

When the system chooses a certain *factor link* leaving from i and labeled by σ_k, the value of $\rho_{i,\sigma}$ is decremented, multiplying it by β. Subsequently, we update the new value of sum_i by subtracting $(1 - \beta) * a_{i,\sigma_k}$ (fig. 8). That way, we preserve the property $\sum_{\sigma \in from_i} [\rho_{i,\sigma}/sum_i] = 1$ for each state i. Note that we are only adding constant time operations, making our model compatible with soft real-time.

Following only the *factor links* we obtain all the factor (subsequences) of the original sequence. This causes two problems: first, if we always follow the *factor links*, soon we will get to the last state of the automaton; second, we only improvise over the subsequences of the information learned from the user, without sequence variation. This would make the improvisation repetitive. Following the *suffix link* we achieve sequence variation because we can combine different suffixes and prefixes of the sequences learned. For instance, in *Omax* [3] –a model for music improvisation processing in real-time audio and video– this is called recombination and it is parametrized by a recombination factor.

Rueda et al approaches this problem in *Ccfomi* by creating a probability distribution parameterized by a value α. The probability of choosing a *factor link* is given by α and the probability of choosing a *suffix link* is given by $1 - \alpha$. There is a drawback in this approach. Since it does not reward the *suffix links* with the *context* (the length of the common suffix), this system may choose multiple times in a row *suffix links* going back one or two states, creating repetitive sequences.

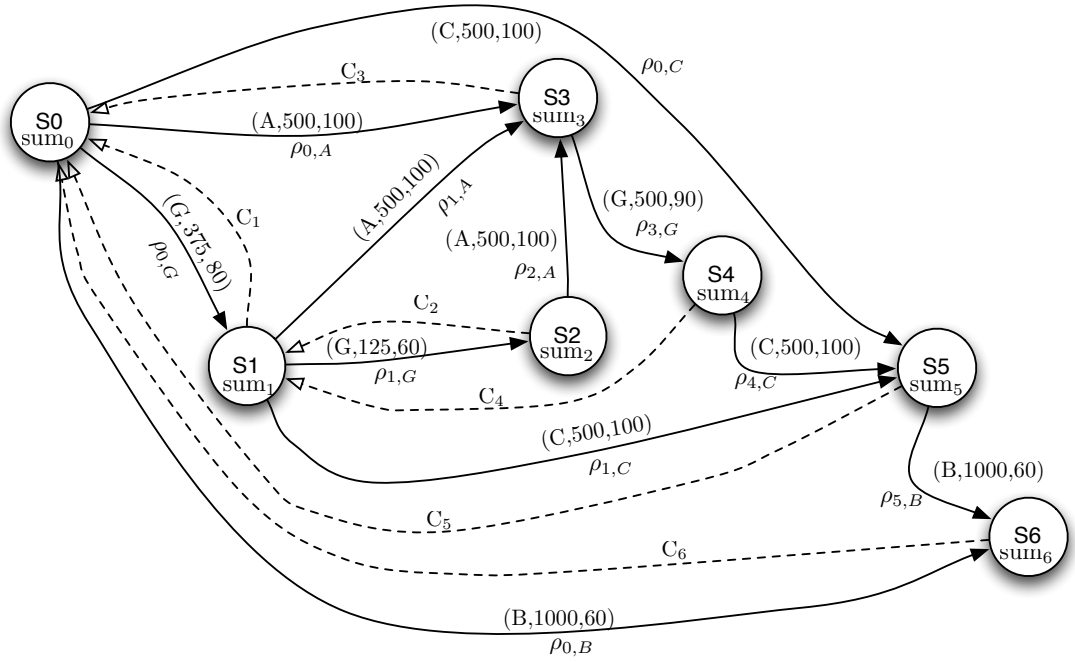
Our approach is based on rewarding the *suffix links* by their *context*. The intuition is choosing between the *factor links* leaving a state i and the *factor links* leaving the state reached by following the



(a) The Score of the fragment

(G, 375,80), (G, 125,60), (A, 500,100), (G, 500,90), (C, 500,100), (B, 1000,60)

(b) Fragment of the Happy Birthday Song represented with tuples



(c) Factor Oracle with the probabilistic information

Figure 7: Factor Oracle used to represent a Happy Birthday fragment

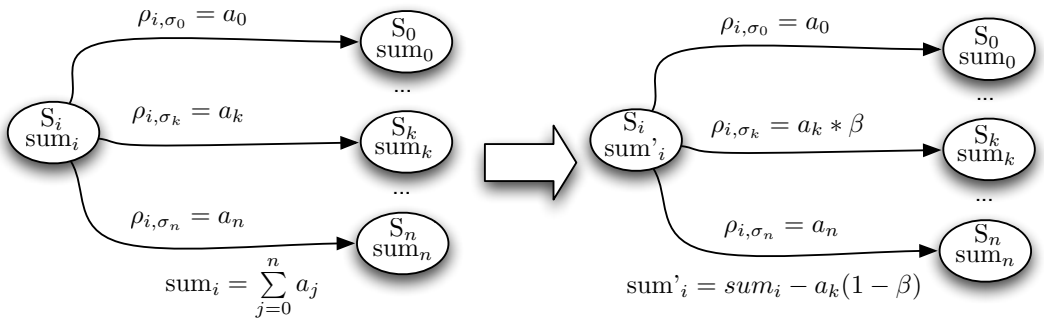


Figure 8: Choosing a *factor link* from k labelled by σ_k

current state's *suffix link*. Rewarding the last ones by the product of the recombination factor α and the *context* $\mathbb{C}_i$. Consider $S(i)$ a function returning the state where a *suffix link* leads from a state i. If we only consider the *factor links*, we would have two probability distributions $\varphi(i, \sigma) = \rho_{i,\sigma}/sum_i$ and

$\varphi(S(i),\sigma) = \rho_{S(i),\sigma}/sum_{S(i)}$ and no way to relate them. Using the *context* $\mathbb{C}_i$, we create a probability distribution $\Phi(i,\sigma)$ ranking the *factor links* leaving from the state $S(i)$ with the product $\alpha * \mathbb{C}(i)$.

$$\Phi(i,\sigma) = \begin{cases} \frac{\rho_{i,\sigma}}{sum_i} & \text{if } S(i) = -1, \\ \frac{\rho_{i,\sigma}+\rho_{S(i),\sigma}*\mathbb{C}_i*\alpha}{sum_i+sum_{S(i)}*\mathbb{C}_i*\alpha} & \text{if } S(i) > -1 \end{cases}$$

Using $\Phi(i,\sigma)$, the system is able to choose a symbol at any state of the FO. The advantage of this probability distribution over the one presented in $Ccfomi$, is that it takes into account the *context*, as well as the recombination factor α.

To exemplify how to build this probability distribution, consider the FO with the probabilistic information in figure 9. That example correspond to the FO for $s = ab$ and random values for the integer arrays described in this chapter. Table 4 shows how to build a probability distribution $\Phi(i,\sigma)$ for the FO in figure 9.

Note that for the states zero and two in the table, the probabilities calculated are the same. This happens because the first state does not have a *suffix link* to go backwards and the last state does not have *factor links* to go forward. On the other hand, the probabilities calculated for the state one combine the probability of choosing a *factor link* following state 1 or choosing the *suffix link* and then choosing a *factor link* from state zero.

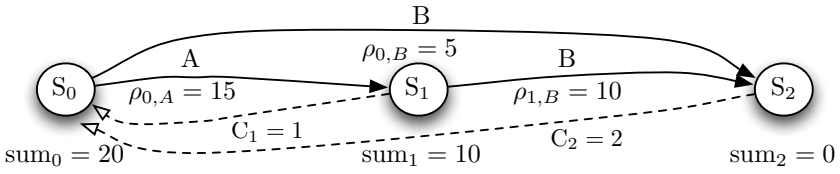


Figure 9: A Factor Oracle including probabilities, for the sequence $s = ab$

i	σ	$\Phi(i,\sigma)$	i	σ	$\Phi(i,\sigma)$	i	σ	$\Phi(i,\sigma)$
0	a	3/4	1	a	$\frac{0+15*\alpha}{10+20*\alpha}$	2	a	3/4
0	b	1/4	1	b	$\frac{10+5*\alpha}{10+20*\alpha}$	2	b	1/4

Table 4: Probability distribution $\Phi(i,\sigma)$ for figure 9

3.3 Summary

In this chapter we explained how we can model music improvisation using probabilities, extending the notion of non-deterministic choice described in $Ccfomi$. The intuition is decreasing the probability of choosing a *factor link*, each time it is chosen and rewarding a *suffix link* based on the *context*. Furthermore, we explained how the parameters α, β, and γ allow us to parameterize the computation of the probabilities.

This procedure is simple enough so that the probabilities can be computed in constant time when the FO is built, preserving the linear complexity in time and space of the FO on-line construction algorithm. Additionally, using probabilities allows us to generate different sequences, without repeating the same sequence multiple times in a row like $Ccfomi$.

3.4 Related work

For $Omax$, Assayag and Blonch recently proposed a new way to traverse the oracle based on heuristics [2]. They argue that traversing the oracle using only the *suffix links* and not using the *factor links*,

produces more "interesting" sequences.

There is an extension of $Ccfomi$ using `pntcc`. The use of `pntcc` makes possible to choose the sequences in the simulation phase, based on a probability distribution. Although Perez and Rueda modeled the probabilistic choice of sequences using the FO, they do not provide a description of how those probabilities can be calculated during the learning phase.

4 Changing Musical Attributes of the Notes

According to Conklin [4], music-generation systems aim to create music based on some predefined rules and a *corpus* (i.e., a collection of musical pieces in a certain music style) learned previously. Those systems can create new musical material based on the style of the *corpus* learned. Unfortunately, they use algorithms with high complexity in time and space, making them inappropriate for music interaction according to [10]. On the other hand, interactive systems for music improvisation (e.g., $Ccfomi$) are usually based on the recombination of sequences learned from the user.

Although recombination creates new sequences based on the user style, it does not create new notes. In fact, it does not even change a single characteristic of a note. To solve that problem, one of the objectives of this article is changing at least one musical attribute of the notes generated during the style simulation.

In the beginning of this work, we tried to develop an algorithm for creating new notes, based on the learned style. The idea was calculating the probability of being on a certain music scale. Based on that probability, we choose a random pitch from that scale. A music scale is an ascending or descending series of notes or pitches. We also developed an algorithm to calculate the duration of those new notes (see Appendix **??**).

We did not include those ideas in this article. First, because choosing a *pitch* based on a supposition of the scale cannot be generalized to music which is not based on scales. In addition, because the procedure for calculating the probability of being on a certain scale was not very accurate, as we found out during some tests. Finally, because the algorithm to generate new durations is not compatible with soft real-time.

The approach we chose to change a musical attribute is again based on simple, but powerful concept. We store the average *intensity* (the other musical attributes are not changed in our model for the reasons mentioned above) of the notes currently being played (*current dynamics*) by the computer. We also store the *current dynamics* of the user. Then, we compare them and change the *current dynamics* of the computer (if necessary), making it similar to the user *current dynamics*. The idea behind this *intensity* variation was originally proposed by musicians Riascos and Collazos . It is based on a concept that they usually apply when improvising with other musicians.

In order to formalize that concept, we calculate, in the learning phase, the *current dynamics* of the last τ (a system parameter) notes played by both, the user and the computer, separately. Concurrently, in the simulation phase, we compare the two *current dynamics*. If they are not equal, we multiply the intensity of the current note being played by the computer by a factor proportional to relation of the user and computer *current dynamics*. As follows, we explain in detail how we can calculate the *current dynamics* in the learning phase and how to change the *intensity* of notes generated in simulation phase.

4.1 Stylistic learning phase

The *intensity* in music represents two different things at the same time. When analyzing the *intensity* of a single note in a sequence, we reason about that *intensity* as a musical accent meaning the importance of certain notes or defining rhythms. On the other hand, we reason about the *average intensity* of a sequence of notes as the dynamics of that sequence of notes. The accents may be written explicitly in the score with a symbol $>$ bellow the note and the dynamics for relative loudness may be written explicitly in the score as piano (p), forte (f), fortissimo (ff), etc.

To capture these two concepts, in the learning phase we store the *intensity* in a tuple $<$ *pitch*, *duration*, *intensity* $>$. In addition, we store the *current dynamics* for the last τ notes played by the user $\overline{Q_u}$ and the computer $\overline{Q_c}$.

To calculate the *current dynamic* we propose the Calculate-Current-Dynamics algorithm. The idea of this algorithm is storing the last τ intensities in a queue Q_i. This algorithm receives a sequence of

intensities I, the value for τ, a reference to the queue Q_i, and the *current dynamic* $\overline{Q_i}$. The invariant of the algorithm is always having the average of the queue data in the variable $\overline{Q_i}$ and the sum in the variable $IntensitySum$. Append **??** gives an example of the operation of this algorithm.

CALCULATE-CURRENT-DYNAMICS($I = I_1 I_2 ... I_m$, τ, $\overline{Q_i}$, Q_i)
01 $Q_i \leftarrow$ new Queue(τ)
02 $IntensitySum \leftarrow 0$
03 QueueSize $\leftarrow 0$
04 **for** $i \leftarrow 0$ **to** m **do**
05 **if** $QueueSize < \tau$ **then**
06 $IntensitySum \leftarrow IntensitySum + I_i$
07 **else** $IntensitySum \leftarrow IntensitySum + I_i$ - $Q_i.pop()$
08 Q_i.push(I_i)
09 $\overline{Q_i} \leftarrow IntensitySum/QueueSize$

4.2 Stylistic simulation phase

In this phase, we traverse the FO using the probabilistic distribution $\Phi(i, \sigma)$ proposed in chapter 3. Remember that there is an *intensity* and a *duration* associated to each *pitch* in the FO. If we play the *intensities* with the same value as they were learned, we could have a problem of coherence between the *current dynamics* of the user and the *current dynamics* of the sequences we are producing.

To give an example of this problem, consider the Happy Birthday fragment presented in figure 7. The *current dynamics* for that fragment is 98. Now, suppose the computer *current dynamics* is 30. This poses a problem, because the user is expecting the computer to improvise in the same *dynamics* that he is, according to the interviews with Riascos and Collazos.

The solution we propose is multiplying by a factor $\overline{Q_c}/\overline{Q_u}$ the intensity of every note generated by the computer. In the previous example, the next note generated by the computer would be multiplied by a factor of 30/98.

4.3 Summary

We explained how we can change the *intensity* of the notes generated during the improvisation. The idea is to maintain the *current dynamics* of the notes generated by the computer similar the *current dynamics* of the notes generated by the user. This corresponds to formalizing an improvisation technique used by two musicians interviewed for this article.

This kind of variation in the *intensity* is something new for machine improvisation systems as far as we know. We believe that this kind of approach, where simple variations can be made preserving the style learned from the user and being compatible with real-time, should be a topic of investigation in future works.

4.4 Related work

To solve this problem of creating new notes and changing the attributes of the notes during the improvisation, the *Omax* model has a parameter called *innovation rate*, indicating the amount of new material desired [3]. Furthermore, Omax calculates a *rhythmic quality function* to compare the density (the number of events for overall duration) between the current state and the place where a link is leading.

Using that *rhythmic quality function*, the improvisation does not "jump" abruptly between different rhythmic patterns. Therefore, *Omax* improvisation is rhythmically coherent within itself. However, generating new rhythms coherent with the user style on machine improvisation is still an open problem.

The anticipatory model developed by Cont et al [6] presents some results where the sequences produced in the improvisation have different pitches, compared to the original sequence. To achieve this, they improvise on a *pitch intervals FO* (a FO learning the intervals of the pitches played by the user), allowing them to calculate new pitches, when using the *pitch intervals* attribute to improvise.

Neither *Ccfomi* nor its probabilistic extension provides a way to change musical attributes of the notes nor creating new material based on the user style.

5 Modeling the system in pntcc

Ntcc has been used in a large variety of situations for synchronizing musical processes. From the introduction chapter, we recall the models for interactive scores, audio processing, formalizing musical processes, and music improvisation. In those models, the synchronization is made declaratively. It means that ntcc hides the details on how the processes are synchronized and how the shared resources (in the *store*) are accessed.

One objective of this work is modeling our improvisation system with ntcc. So far, we presented the modifications for the improvisation phases allowing probabilistic choice of musical sequences and changing the musical attributes in the simulation phase. Since we are choosing the sequences probabilistically, we use pntcc (the probabilistic extension of ntcc) for modeling our improvisation system.

In order to synchronize the improvisation phases, the learning phase must take place from the beginning. However, the simulation phase is launched once the learning phase has learned n notes. After that, both phases run concurrently. Synchronization must be provided because the improvisation phase must not work in partially built graphs, it can only improvise in the fragment of the graph that represents a FO. Additionally, the simulation phase can only work in state k once the value for the *current dynamics*, the *context*, and the *probabilistic distribution* has been calculated up to state k.

Our approach to synchronize the improvisation phases is similar to the one used in *Ccfomi*. Remember that *Ccfomi* synchronizes the improvisation phases using a variable *go* and the variables S_i. The $PLAYER$ process can post constraints over those variables and the processes for building the FO (ADD and $LOOP$) are activated when they can deduce certain information from those variables. We extend that concept using some of the new variables introduced in this model.

In addition to the variables $from_k$, S_i, and $\delta_{i,\sigma}$ used in $Ccfomi$, our model has a few more variables: $\rho_{i,\sigma}$, sum_i, $\Phi_{i,\sigma}$, and $\mathbb{C}_i$ represent the probabilistic choice of musical sequences; $duration_\sigma$ and $intensity_\sigma$ represent the musical attributes associated to each pitch σ; and Q_i, $SumQ_i$ and $\overline{Q_i}$ represent the intensity variation. The variables $\Phi_{i,\sigma}$, $\mathbb{C}_i$, Q_i, $duration_\sigma$, and $intensity_\sigma$ are represented with rational trees of FD variables because they do not change their value from a *time-unit* to another. The other variables are represented with cells (cells are defined in chapter 2).

In this chapter, we explain how we can write a sequential algorithm for the learning phase combining the algorithm for building on-line the FO, calculating the *context*, calculating the probabilistic distribution $\Phi_{k,\sigma}$ and the *current dynamics*. After that, we show how both phases can be modeled in pntcc. Finally, we give some concluding remarks and we present related work.

5.1 Modeling the stylistic learning phase

The learning phase can be easily integrated to the on-line algorithm that builds a FO and calculates the *context* (the original algorithms are presented in Appendix **??**). The learning phase is represented by the functions *Ext_Oracle_On-line* and *Ext_Add_Letter*. To calculate the *context* we use the *Length_Repeated_Suffix* function proposed by Lefevre et al. The *Length_Repeated_Suffix* calculates the *context*. It finds the length of a repeated suffix of $P[1..i+1]$ in linear time and space complexity.

The *Ext_Add_Letter* function is in charge of adding new pitches to the FO. It also creates a tuple $< pitch, duration, intensity >$; updates values of $\rho_{i,\sigma}$ and sum_i; and calculates the *current dynamics* of the user $\overline{Q_u}$, and the *context* $\mathbb{C}_{i+1}$ for state $i+1$. This function receives a FO with i states, a pitch σ, the duration, the intensity, the system parameters γ and τ, and the Intensity Queue Q_i. During its execution, it uses the constant c, the function $S(i)$, and the temporal variable π. C is a big integer constant, $S(i)$ is a function returning the *suffix link* for state i, and π is a temporal variable used to calculate the *context*.

EXT_ADD_LETTER(Oracle($P[1..i]$),σ,$duration$,$intensity$,γ,τ,Q_i)
01 Create a new state $i+1$
02 $\delta(i,\sigma) \leftarrow i+1$
03 $k \leftarrow S(i)$

04 $\pi_1 \leftarrow i$
05 $\rho_{i,\sigma} \leftarrow c$
06 $sum_i \leftarrow c$
07 **if** $QueueSize < \tau$ **then**
08 $intensitySum \leftarrow IntensitySum + intensity$
09 **else** $IntensitySum \leftarrow intensity$ - $Q_i.pop()$
10 Q_i.push(I_i)
11 $\overline{Q_i} \leftarrow IntensitySum/QueueSize$
12 $duration_\sigma \leftarrow duration$
13 $intensity_\sigma \leftarrow intensity$
14 **While** $k > -1$ and $\delta(k, \sigma)$ is undefined **do**
15 $\delta(k, \sigma) \leftarrow i + 1$
16 $\pi_1 \leftarrow k$
17 $k \leftarrow S(k)$
18 $\rho_{k,\sigma} \leftarrow sum_k * \gamma$
19 $sum_k \leftarrow sum_k * (1 + \gamma)$
20 **if** $k = -1$ **then** $S(i + 1) \leftarrow 0$
21 **else** $S(i + 1) \leftarrow \delta(k, \sigma)$
22 $\mathbb{C}_{i+1} \leftarrow LENGTH\_REPEATED\_SUFFIX(\pi_1, S(i + 1))$
23 **Return** Oracle(P[1..i]σ)

The *Ext_Oracle_On-line* function is a sequential algorithm representing the learning phase. It receives three vectors: the pitches, the durations, and the intensities. In addition, it takes γ, the system parameter for ranking the importance of a new note added to the FO, and the system parameter τ, representing the number of notes taken into account to calculate the *current dynamics* . Figure 10 presents the execution of this function for the three first symbols of the Happy Birthday Fragment presented in figure 7.

EXT_ORACLE_ON_LINE(P[1..m],D[1..m],I[1..m],γ,τ)
01 Create Oracle(ϵ) with one single state 0 and $S(0) = -1$
02 $Q_u \leftarrow new\ Queue(\tau)$
03 $IntensitySum \leftarrow 0$
04 **for** $i \leftarrow 1$ **to** m **do** $Oracle([1..i]) \leftarrow$
05 **EXT_ADD_LETTER**(Oracle(P[1..$i-1$]),P_i,D_i,I_i,γ,τ,Q_u)

The learning phase is modeled in `pntcc` by the processes PHI, ADD, $LOOP$, $ADD\_ELEMENT$, $PLAYER$, $CONTEXT$, and $CONTEXT\_LOOP$. Process PHI calculates the values for the probability distribution $\phi_{k,\sigma}$, used to choose A *factor link* leaving from state k labeled by a symbol σ. Where the recombination factor is parameterized by α. The process $Tree|_-$ represents the act of adding a "fresh" variable to the infinite rational tree (as described in chapter 2). We use infinite rational trees to represent the variable such as $from$ and δ that represents the transitions of the FO.

$PHI(k, \sigma, \alpha) \stackrel{def}{=}$
when $S_k = -1$ **do !tell** $(\phi_{k,\sigma} = \frac{\rho_{k,\sigma}}{sum_k})$
$\|$**when** $S_k > -1$ **do !tell** $(\phi_{k,\sigma} = \frac{\rho_{i,\sigma} + \rho_{S(i),\sigma} * \mathbb{C}_i * \alpha}{sum_i + sum_{S(i)} * \mathbb{C}_i * \alpha})$ $\|$ $\phi|_-$

The Process ADD is the one in charge of adding new pitches to the FO. In addition, this process updates the values of the cells ρ and the variable ϕ calling the function PHI.

$ADD_i(\alpha, \gamma) \stackrel{def}{=}$
$\delta_{i-1,\sigma_i} \leftarrow i$ $\|$ **!tell** $(\sigma_i \in from_{i-1})$ $\|$ $\rho_{i-1,\sigma_i} : (c)$ $\|$ $sum_{i-1} : (c)$
$\|$ $PHI(i - 1, \sigma_i, \alpha)$ $\|$ $from|_-$ $\|$ $\delta|_-$ $\|$ $\rho|_-$ $\|$ $sum|_-$ $\|$ **next** $LOOP_i(S_{i-1}, \alpha, \gamma, i - 1)$

The $LOOP$ process represents the "while" loop in the *Ext_Add_Letter* function. This process adds a new *factor link* in the FO that points to the new state i, while k is greater than -1 and there is not a

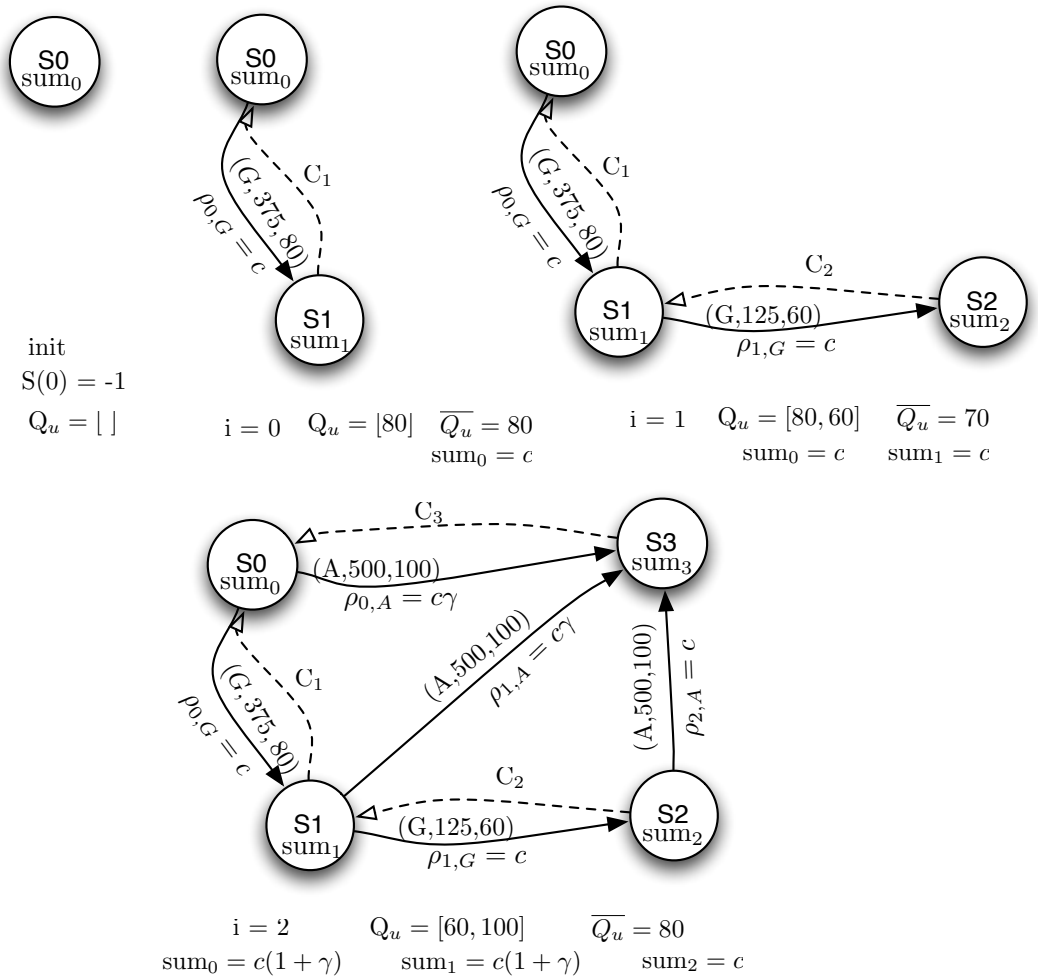


Figure 10: Executing the *Ext Oracle On-line* algorithm with $\tau = 2$

transition from k labeled by σ. The values for k depends on the *suffix links*. In addition, it calculates the values for the *context* $\mathbb{C}_i$ and the probabilistic information.

$LOOP_i(k, \alpha, \gamma, \pi_1) \stackrel{def}{=}$
when $k \geq 0$ **do**(
when $\sigma_i \in from_k$ **do**
(**!tell** $(S_i = \delta_{k,\sigma_i}) \parallel S|_{\text{-}} \parallel CONTEXT(i, \pi_1, S_i)$)
$\|$**unless** $\sigma_i \in from_k$ **next** (
$sum_k :\leftarrow sum_k(1+\gamma) \parallel sum|_{\text{-}} \parallel \rho_{k,\sigma_i} :\leftarrow \gamma sum_k \parallel \rho|_{\text{-}} \parallel PHI(\alpha, k, \sigma_i)$
$\parallel$ **next** (**!tell** $(\sigma_i \in from_k) \parallel$ **!tell** $(\delta_{k,\sigma_i} = i)$
$\parallel LOOP_i(S_k, \alpha, \gamma, k) \parallel from|_{\text{-}} \parallel \delta|_{\text{-}})))$
$\|$**when** $k = -1$ **do** (**!tell** $(S_i = 0) \parallel S|_{\text{-}} \parallel CONTEXT(i, \pi_1, S_i)$)

In the $CONTEXT$ process the reader may notice how we can use **when** $a \neq b$ **do** P instead of **unless** $a \neq b$ **next** P because we know that a,b always have a value. The values π, s, π_1 and pi_2 are used to calculate efficiently the *context* according to Lefevre et al.'s algorithm.

$CONTEXT(i, \pi, s) \stackrel{def}{=}$

when $s = 0$ **do** **!tell** $(\mathbb{C}_i = 0)$
$\|$**when** $s \neq 0$ **do** (
when $s - 1 = S_{\pi 1}$ **do** **!tell** $(\mathbb{C}_i = C_{\pi_1} + 1)$
$\|$**when** $s - 1 \neq S_{\pi 1}$ **do** $CONTEXT\_LOOP(\pi, s - 1, i))$

$CONTEXT\_LOOP(\pi_1, \pi_2, i) \stackrel{def}{=}$
when $S_{\pi_1} = S_{\pi_2}$ **do** **!tell** $(\mathbb{C}_i = min(\mathbb{C}_{\pi_1}, \mathbb{C}_{\pi_2}) + 1)$
$\|$**when** $S_{\pi_1} \neq S_{\pi_2}$ **do next** $CONTEXT\_LOOP(\pi_1, S_{\pi_2}, i)$

The $ADD\_ELEMENT$ process calculates the value for the *current dynamics*. In addition, it updates *sum* based on the parameter τ.

$ADD\_ELEMENT(Q, I, \tau, index, sum, \overline{Q}) \stackrel{def}{=}$
when $index \geq \tau$ **do** $sum :\leftarrow sum + I - Q_{index-\tau}$
$\|$**when** $index < \tau$ **do** $sum :\leftarrow sum + I \quad \| \ \overline{Q} :\leftarrow sum/min(index, \tau)$

Finally, the $PLAYER$ stores the values of *pitch*, *duration*, and *intensity* received from the environment when a note is played by the user. Furthermore, it updates the *current dynamics* $\overline{Q_u}$.

$PLAYER_j \stackrel{def}{=}$
when $P > 0 \wedge D > 0 \wedge I > 0$ **do** (
$ADD\_ELEMENT(Q_u, I, \tau, j, SumQ_u, \overline{Q_u})$
$\|$ **next** (**!tell** $(\sigma_j = P) \ \| \ Q_u|_{\_} \ \|$ **!tell** $Qu_j = I$
$\|$ **!tell** $(duration_{\sigma_j} = D) \ \|$ **!tell** $(intensity_{\sigma_j} = I) \ \|$ **tell** $(go = i) \ \| \ PLAYER_{j+1}))$
$\|$**unless** $P > 0 \wedge D > 0 \wedge I > 0$ **next** (**tell** $(go = j - 1) \ \| \ PLAYER_j$)

5.2 Modeling the style simulation phase

In this phase, we use the $\bigoplus$ agent, defined in `pntcc` to model probabilistic choice. This model is an extension of the model presented in [20]. In our model, the $IMPROV$ process chooses a link according to the probability distribution ϕ_{k,σ_i}. Furthermore, it updates the values for *sum* and ρ, sets-up the outputs, and updates the computer *current dynamics* $\overline{Q_c}$.

In order to ask if a constraint $A \wedge B$ or $A \vee B$ can be deduced from the *store*, we use reification. For instance, the process **when** $a = b \wedge c = d$ **do** P, can be codified as the process **when** g **do** P and the constraints $a = b \leftrightarrow e$, $c = d \leftrightarrow f$, $e \wedge f \leftrightarrow g$.

$IMPROV(k, \tau, \beta) \stackrel{def}{=}$
$\|$ **when** $C_k \geq 0$ **do** (
$\bigoplus_{i \in \sum}$ **when** $\sigma_i \in from_k \vee \sigma_i \in from_{S_k}$ **do** (
next(**tell** $(out\_pitch = \sigma_i) \ \|$ **tell** $(out\_duration = duration_{\sigma_i})$
$\|$ **tell** $(out\_intensity = intensity_{\sigma_i} * \overline{Q_u}/\overline{Q_c}) \ \|$ **tell** $(Qc_i = intensity_{\sigma_i})$
$\|$ $ADD\_ELEMENT(Q_c, out\_intensity, \tau, i, SumQ_c, \overline{Q_c}))$
$\|$ **when** $\sigma_i \in from_k \wedge \sigma_i \in from_{S_k}$ **do next** $(IMPROV(k + 1, \tau, \beta) + IMPROV(S_k, \tau, \beta))$
$\|$ **unless** $\sigma_i \in from_k$ **next** ($IMPROV(S_k, \tau, \beta)$
$\| \ \rho_{S_k,\sigma_i} :\leftarrow \beta * \rho_{S_k,\sigma_i} \ \| \ sum_{S_k} :\leftarrow sum_{S_k} - (1 - \beta)\rho_{S_k,\sigma_i}$)
$\|$ **unless** $\sigma_i \in from_{S_k}$ **next** ($IMPROV(k + 1, \tau, \beta)$
$\| \ \rho_{k,\sigma_i} :\leftarrow \beta * \rho_{k,\sigma_i} \ \| \ sum_k :\leftarrow sum_k - (1 - \beta)\rho_{k,\sigma_i}$),$\Phi_{k,\sigma_i})$
$\|$ **unless** $C_k \geq 0$ **next** $IMPROV(k, \tau, \beta)$

5.3 Synchronizing the improvisation phases

Synchronizing both phases is greatly simplified by the used of constraints. When a variable has no value, *when* processes depending on it are blocked. Therefore, the $SYNC_i$ process is "waiting" until *go* is

greater or equal than one. That means that the $PLAYER_i$ process has played the note i and the ADD_i process can add a new symbol to the FO. The other condition $S_{i-1} \geq -1$ is because the first *suffix link* of the FO is equal -1 and that *suffix link* cannot be followed in the simulation phase. In addition, the $SYNC$ process is also "waiting" for the *current dynamics* $\overline{Q_u}$ to take a value greater of equal than 0.

$SYNC_i(\alpha, \gamma) \stackrel{def}{=}$
when $S_{i-1} \geq -1 \wedge go \geq i \wedge \overline{Q_u} > 0$ **do** $(ADD_i(\alpha, \gamma) \parallel$ **next** $SYNC_{i+1}(\alpha, \gamma))$
$||$**unless** $S_{i-1} \geq -1 \wedge go \geq i \wedge \overline{Q_u} > 0$ **next** $SYNC_i(\alpha, \gamma)$

A $wait_n$ process is necessary to wait until n symbols have been learned to launch the $IMPROV$ process.

$WAIT(n, \tau, \beta) \stackrel{def}{=}$
when $go = n$ **do next** $IMPROV(n, \tau, \beta)$ $||$ **unless** $go = n$ **next** $WAIT(n, \tau, \beta)$

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 start the $IMPROV$ process. The reader should remeber that α is the recombination factor, representing the proportion of new sequences desired. β represents the factor for decreasing the importance of a *factor link* when it is chosen in the simulation phase. γ represents the importance of a new *factor link* in relation with the other *factor links* coming from the same state. τ is a parameter for changing musical attributes in the notes. Finally, n is a parameter representing the number of notes that must be learned before starting the simulation phase.

$SYSTEM(n, \alpha, \beta, \gamma, \tau) \stackrel{def}{=}$
!tell $(S_0 = -1) \parallel SYNC_1(\alpha, \gamma) \parallel PLAYER_1(\tau) \parallel WAIT(n, \tau, \beta)$

5.4 Summary

We modeled all the concepts described in previous chapters using `pntcc`. Although synchronization and probabilistic choice are modeled declaratively, matching the *time-units* is not an easy task because the value of a cell only can be changed in the following *time-unit*. If we change the value of a cell in the scope of an *unless* process, we need to be aware that the value will only be changed two *time-units* after.

5.5 Related work

The *Omax* model uses *FO*, but instead of using `ntcc`, it uses shared state concurrency (for synchronizing the improvisation phases) and message passing concurrency (for synchronizing OpenMusic and Max/Msp). Although this a remarkable model, we believe that `ntcc` can provide an easier way to synchronize processes and to reason about the correctness of the implementation because it is obviously easier to synchronize declaratively. Constraints provide a much more powerful way to express declaratively complex synchronizing patterns. Since the `ntcc` model has a logical counterpart [17], it is possible to prove properties of the model. For instance, the fact that it always (or never or sometimes) chooses the longest context, or that repetitions of some given subsequence are avoided.

Probabilistic Ccofmi [20] fixes the problems with synchronization and extends the notion of probabilistic choice in the improvisation phase, giving it a clear and concise semantic. However, it does not model how can probabilistic distributions may change from a *time-unit* to another based on user and computer interaction.

6 Implementation

A `ntcc` interpreter is a program that takes `ntcc` models and creates a program that interacts with an environment, simulating the behavior of the `ntcc` models. `Ntcc` interpreters (including our interpreter)

are designed to simulate a finite `ntcc` model. It means that they only simulate a finite number of *time-units*.

During the last decade, three interpreters for `ntcc` have been developed. *Lman* [16] by Hurtado and Muñoz in 2003, *NtccSim* (http://avispa.puj.edu.co) by the Avispa research group in 2006, and *Rueda's sim* in 2006. They were intended to simulate `ntcc` models, but they were not made for real-time interaction. Recall from the introduction that soft real-time interaction means that the user does not experience noticeable delays in the interaction.

When designing a `ntcc` interpreter, we need a constraint solving library or programming language allowing us to check stability (i.e., know when a *time-unit* is over), check entailment (i.e., know if a constraint can be deduced from the *store*), post constraints, and synchronize the concurrent access to the *store*. These tasks must be performed efficiently to achieve a good performance.

The authors of the `ntcc` model for interactive scores proposed to use Gecode as a constraint solving library for future `ntcc` interpreters, and create an interface for Gecode to OpenMusic to specify multimedia interaction applications. Furthermore, they proposed to extend *Lman* to work under Mac OS X using Gecode.

One objective of this article is to develop a prototype for a `ntcc` interpreter real-time capable. We followed the advise from the authors of the interactive scores model and we tried out several alternatives to develop an interpreter using Gecode.

Our first attempt was using a thread to represent each `ntcc` process in the simulation. However, we found out that using threads adds an overhead in the performance of the interpreter because of the context-switch among threads, even when using lightweight (lw) threads. Then, we tried using event-driven programming. Performance was better compared with threaded implementations. However, each time a **when** process asks if a condition can be entailed, we need to check for stability, thus adding an unnecessary overhead. The reader may find more information about our previous attempts in Appendix **??** and performance results in chapter 7.

Our implementation, *Ntccrt*, is once again based on a simple but powerful concept. The **when** and $\sum$ processes are encoded as propagators in Gecode. That way Gecode manages all the concurrency required for the interpreter. Gecode calls the continuation of a process when a process condition is assigned to true.

On the other hand, **tell** processes are trivially codified to existing Gecode propagators and timed agents (i.e. $*$, !, **unless**, $\leftarrow$ and **next**) are managed providing different process queues for each *time-unit* in the simulation.

Our interpreter works in two modes, the developing mode and the interaction mode. In the developing mode, the users may specify the `ntcc` system that they want to simulate in the interpreter. In the interaction mode, the users execute the models and interact with them.

This chapter is about the design and implementation of *Ntccrt*. We explain how to encode all the `ntcc` processes. We also explain the execution model of the interpreter. After that, we show how to run *Ccfomi* in the interpreter.

In addition, we describe how we made an interface to OpenMusic and how we can generate binary plugins for data-flow programming languages: Pure Data (Pd) [22] or Max/Msp [23] where MIDI, audio, or video inputs/outputs can interact with a *Ntccrt* binary. Finally, we give some conclusions, future work, and a short description of the other existing interpreters. A detailed description of *Ntccrt*, the generation of binary plugins, Pure Data, Max/Msp, and the previous *Ntccrt* prototypes can be found in a previous publication [35].

6.1 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++. In addition, currently, it is possible to execute `ntcc` models as a stand-alone program or as an *external* object (i.e., a binary plugin) for Pd or Max.

6.1.1 Developing mode

In order to write a `ntcc` model in *Ntccrt*, the user may write it directly in C++, use a parser that takes Common Lisp macros as input or defining a graphical "patch" in OpenMusic. Using either of these representations, it is possible to generate a stand-alone program or an *external* object (fig 11).

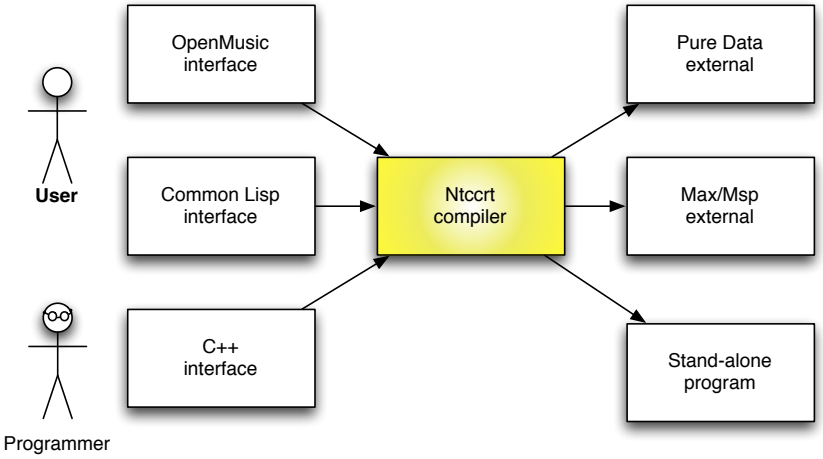


Figure 11: Ntccrt: Developing mode

To make an interface for OpenMusic, first, we developed a Lisp parser using Common Lisp macros to write an `ntcc` model in Lisp syntax and translate it to C++ code. Lisp macros extend Lisp syntax to give special meaning to characters reserved for users for this purpose. Executing those macros automatically compile a `ntcc` program.

After the success with Lisp macros, we created OpenMusic methods to represent `ntcc` processes. Openmusic methods are a graphical representation using the Common Lisp Object System (CLOS). Those graphical objects are placed on a graphical "patch". Executing the "patch" generates a *Ntccrt* C++ program.

6.1.2 Execution mode

To execute a *Ntccrt* program we can proceed in two different ways. We can create a stand-alone program that can interact with the Midishare library [8], or we can create an *external* object for either Pd or Max. An advantage of compiling a `ntcc` model as an *external* object lies in using control signals and the message passing API provided by Pd and Max to synchronize any graphical object with the *Ntccrt* external.

To handle MIDI streams (e.g., MIDI files, MIDI instruments, or MIDI streams from other programs) we use the predefined functions in Pd or Max to process MIDI. Then, we connect the output of those functions to the *Ntccrt* binary plugin. We also provide an interface for Midishare, useful when running stand-alone programs (fig. 12).

6.2 Implementation of *Ntccrt*

Ntccrt is the first `ntcc` interpreter written in C++ using Gecode. In this section, we focus on describing the data structures required to represent each `ntcc` agent. Then, we explain how the interpreter makes a simulation of a `ntcc` model. `Ntcc` agents are represented by classes. To avoid confusions, we write the agents with **bold font** (e.g., **when** C **do** P) and the classes with *italic font* (e.g., *When* class).

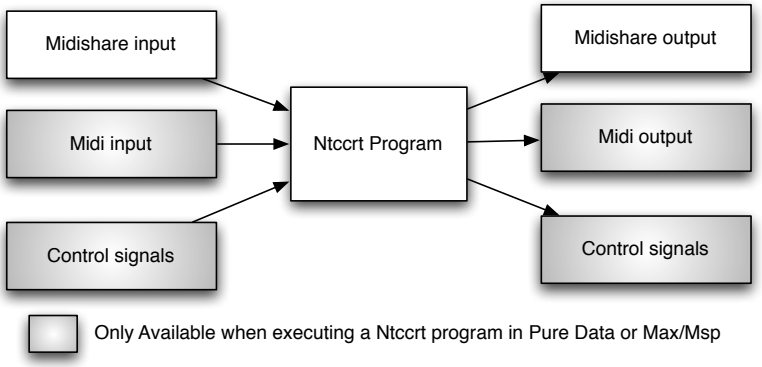


Figure 12: Ntccrt: Interaction mode

6.2.1 Data structures

To represent the constraint systems we need to provide new data types. Gecode variables work on a particular *store*. Therefore, we need an abstraction to represent `ntcc` variables present on multiple *stores* with the same variable object. Boolean variables are represented by the $BoolV$ class, FD variables by the $IntV$ class, FS variables by the $SetV$ class, and infinite rational trees (with unary branching) by $SetVArray$, $BoolVArray$, and $IntVArray$ classes.

After encoding the constraint systems, we defined a way to represent each process. All of them are classes inheriting from *AskBody*. *AskBody* is a class, defining an *Execute* method, which can be called by another object when it is nested on it.

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

For the **when** agent, we made a *When propagator* 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 13 shows how to encode the process **when** $a = c$ **do** P using the *When propagator*

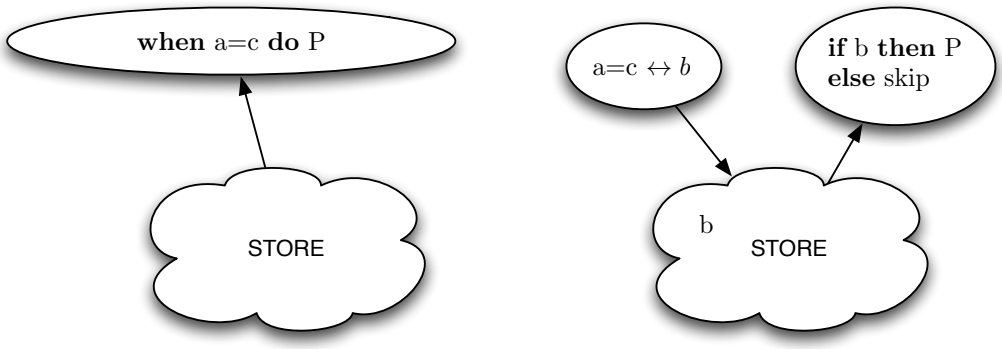


Figure 13: Example of the *When propagator*

To represent the $\sum$ agent (i.e. non-deterministic choice) we made the *parallel conditional propagator*. This propagator receives a sequence of tuples $[< b_1, P_1 > ... < b_n, P_n >]$, where b_i is a *Gecode* boolean

variable representing the condition of a *reified propagator* (e.g., $a = c \leftrightarrow b_i$) and P_i (a pointer to an *AskBody* object) is the process to be executed when b_i is assigned to *true*.

The *When propagator* executes the process P_k associated to the first guard that is assigned to true. It means P_k such that $k = min(\{1 \leq i \leq n, b_i = true\})$. Then, its work is over. If all the variables are assigned to $false$, its work is over too.

The *When propagator* is based on the idea of the *Parallel conditional combinator* proposed by Schulte [33]. A curious reader might ask how we obtain a non-deterministic behavior. In order to make a non-deterministic choice, we pass the parameters to the propagator in a random order. That way, the propagator always chooses the first process which condition is true, but since the processes (and conditions) are given in a random order, it will simulate a non-deterministic choice. Figure 14 shows how to encode the process $\sum x \in \mathbb{P}$ **when** $x \in wait_j$ **do tell** $(control_j = x)$ using the *parallel conditional propagator*. This process is explained in Appendix **??**.

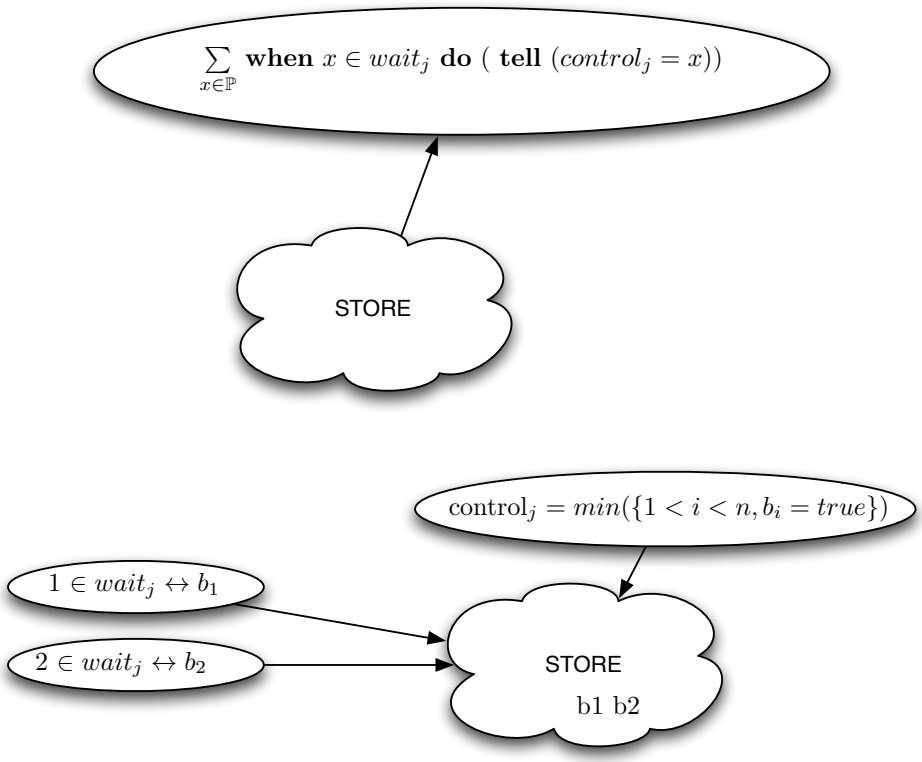


Figure 14: Example of the *Parallel conditional propagator*

Local variables are easily represented by an instruction allowing the user to create a new variable at the beginning of a procedure. Then, that new variable persists during the following *time-units* being simulated. This implementation of local variables is useful when there is a process $!P$ and P contains local variables. The other variables are declared at the beginning of the simulation.

Timed agents are represented by the *TimedProcess* class. *TimedProcess* is an abstract class providing a pointer for the current *time-unit*, for a queue used for the **unless** processes, for a queue used for the persistent assignation processes, for a queue used for the other processes, and for the continuation process. Each subclass defines a different *Execute* method. For instance, the *Execute* method for the *Star* class randomly chooses the *time-unit* to place the continuation (an *AskBody* object) in its corresponding *process queue*.

The *Unless* class and the *Persistent assignation* class are different. The *Execute* method of the *Unless* objects and the the *Persistent assignation* objects are called after calculating a $fixpoint$ common to all

the processes in the *process queue*. Formally, a *propagator* can be seen as a function $F : \mathbb{S} \to \mathbb{S}$, receiving a *store* and returning a *store*. A *fixpoint* for a *propagator* is a *store* x such that $F(x) = x$. When Gecode calculates a *store*, which is a *fixpoint* for all *propagators*, we said that the *store* is stable.

After calculating a $fixpoint$, if the condition for the *Unless* cannot be deduced from the stable *store*, its continuation is executed in the next *time-unit*. On the other hand, the *Persistent assignation* copies the domain $\mathbb{D}$ of the variable assigned, when the *store* is stable. Then, it assigns $\mathbb{D}$ to that variable in following *time-units* (creating a *tell* object for each following *time-unit*).

We also have a *Procedure* class used to model both, `ntcc` simple definitions (e.g., $\mathrm{A} \stackrel{def}{=} \mathbf{tell}(a = 2)$) and `ntcc` recursive definitions (e.g., $\mathrm{B}(i) \stackrel{def}{=} B(i+1)$), which are invocated using the *Call* class. For `ntcc` recursive definitions, we create local variables simulating call-by-value (as it is specified in the formalism). Recursion in `ntcc` is restricted. Parameters have to be variables in the *store* and we can only make a recursive call in a recursive procedure. However, *Ntccrt* does not check these conditions (they are left to the user) and implements general recursion.

6.2.2 Execution model

In order to execute a simulation, the users write a `ntcc` mdel in *Ntccrt*, compile it, and then they call the compiled program with the number of units to be simulated and the parameters (if any) of the main `ntcc` definition. For each *time-unit* i, the interpreter executes the following steps: First, it creates a new *store* and new variables in the *store*. Then, it processes the input (e.g., MIDI data coming from *PD* or *Max*). If it is simulating the first *time-unit*, it calls the main `ntcc` definition with the arguments given by the user.

After that, it moves the *unless* processes to the i^{th} *unless queue*, moves the *persistent assignation* processes to the i^{th} *persistent assignation queue*, and executes all the remaining processes in the i^{th} *process queue*. Then, it calculates a $fixpoint$. Note how we only calculate one $fixpoint$ each *time-unit*, opposed to the previous prototypes.

After calculating a $fixpoint$, it executes the *unless* processes in the i^{th} *unless queue* and executes the *persistent assignations* in the i^{th} *persistent assignation queue*. Then, it calls the *output processing* method (e.g., sending some variable values to the standard output or through a MIDI port). Finally, it deletes the current *store*. Figure 15 illustrates the execution model.

6.3 Implementation of *Ccfomi*

Rueda et al ran *Ccfomi* on their interpreter. They wrote Lisp macros to extend Lisp syntax for the definition of `ntcc` processes. We provide a similar interface to write `ntcc` processes in Lisp. Furthermore, we can write *Ccfomi* definitions in *Ntccrt* in an intuitive way using OpenMusic. For instance, the $Sync_i$ process (presented in chapter 2), in charge of the synchronization between the $PLAYER_i$ and the ADD_i processes, is represented with a few boxes: one for **parallel** processes, one for the $\leq$ condition, one for the $=$ condition, and one for **when** and **unless** processes (fig. 16)

We successfully ran $Ccfomi$ as an stand-alone program using Midishare. We present the results of our tests with the stand-alone program in Chapter **??**. We also ran it as a Pd plugin generated by *Ntccrt*. The plugin is connected to the midi-input, midi-output, and a clock (used for changing from a *time-unit* to the other). For simplicity, we generate a clock pulse for each note played by the user (fig. 17). In the same way, we could connect a $Metronome$ object. $Metronome$ is an object that creates a clock pulse with a fixed interval of time.

6.4 Summary

Further than just developing an interpreter, we developed an interface for OpenMusic to write `ntcc` models for *Ntccrt*. Although the OpenMusic interface generates code for *Ntccrt*, it is not able to embed Lisp code in the interpreter. In addition, the current version of the interpreter does not support probabilistic choice nor cells, required to run our model. This is acceptable because our objective was just to develop a `ntcc` interpreter prototype. For that reason, we still do not support `pntcc` nor cells (which are not basic

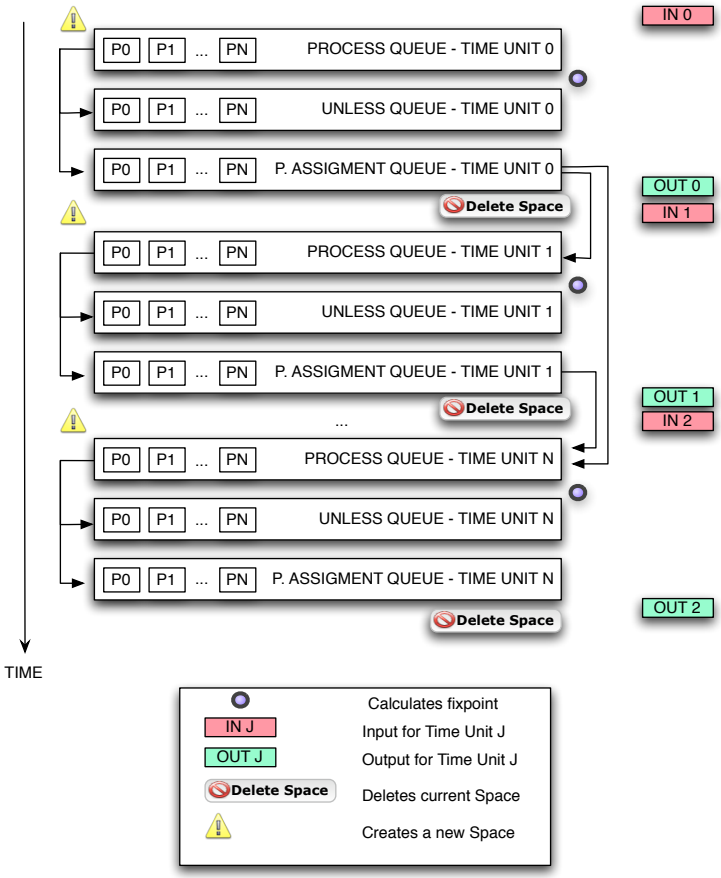


Figure 15: Execution model of the `ntcc` interpreter

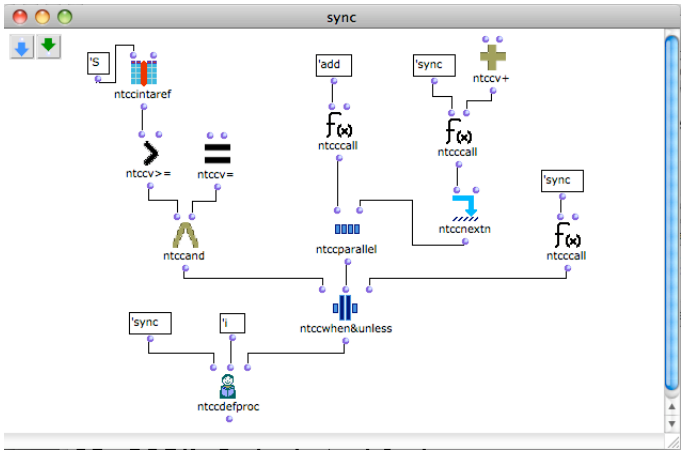


Figure 16: Writing the $Sync_i$ process in OpenMusic

operators on `ntcc`). In the following, we will describe the possibilities and limitations of the interpreter and possible solutions for future work.

Additionally, since we encoded the **When** processes as a *Gecode* propagators, we are able to use

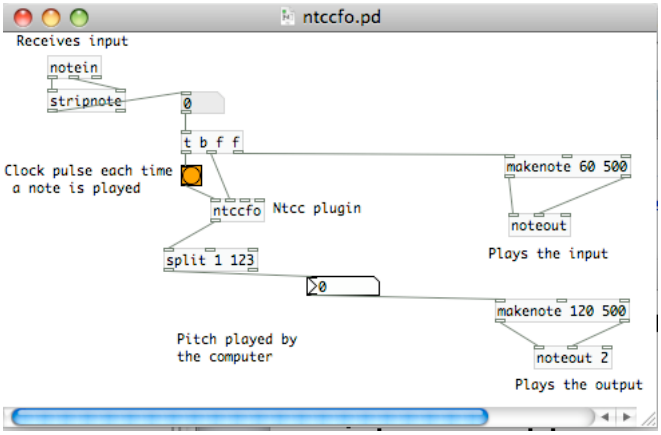


Figure 17: Running *Ccfomi* in Pure Data (Pd)

search in `ntcc` models without using the $\sum$ agent. This is not possible when encoding the **when** processes as lightweight threads or OS threads because threads cannot be managed inside Gecode search engines. Models using non-deterministic choices are incompatible with the recomputation used in the search engines.

Ntccrt cannot simulate processes leading the *Store* to false. For instance,

when $false$ **do next tell** $(fail = true)$
$\|$**tell** $(a = 2)\|$**tell** $(a = 3)$

Since the **when** agent is represented as a propagator, once the propagation achieves a fail state no more propagators will be called in that *time-unit*, causing inconsistencies in the rest of the simulation. Fortunately, processes reasoning about a false *Store* can be rewritten in a different way, avoiding this kind of situations. For instance, the process above can be rewritten as:

when $state = false$ **do next tell** $(fail = true)$
$\|$**tell** $(a = 2)\|$**tell** $(a = 3)\|$**tell** $(state = false)$

Although in many applications we do not want to continue after the *store* fails in a *time-unit* because a failed *store* is like an exception in a programming language (e.g., division by zero).

In addition, *Ntccrt* restricts the domains for the different constraint systems. The domain for FD variables is $[1 - 2^{31}, 2^{31} - 1]$ and each set or tree in the FS and the rational trees variables cannot have more than 2^{31} elements[4]. This limitation is due to Gecode, which uses the C++ integer data type for representing its variables.

Another problem arises when we want to call Lisp functions in the interpreter. This will be usefull to make computer music programs (written in Lisp) to interact with *Ntccrt*. Currently, we are only using Lisp to generate C++ code. However, it is not possible to embed Lisp code in the interpreter (e.g., calling a Lisp function as the continuation of a **when** process). To fix that inconvenient, we propose using *Gelisp* for writing a new interpreter, taking advantage of the call-back functions provided by the Foreign Function Interface (FFI) to call Lisp functions from C++. That way a propagator will be able to call a Lisp function. Although, this could have a negative impact on performance and in the correctness of the system (e.g., when the Lisp function does not end).

The implementation of cells is still experimental and it is not yet usable. The idea for a real-time capable implementation of cells is extending the implementation of persistent assignation. Cells, in the same way than persistent assignation, require to pass the domain of a variable from the current *time-unit* to a future *time-unit*. However, persistent assignation usually involves simple equality relations. On the

[4]It is not 2^{32} because one bit is used for the sign.

other hand, the cells assignation may involve any mathematical function $g(x)$ (e.g $g(x) = x^2 - 2$).

Probabilistic choice is not yet possible neither. For achieving it, we propose extending the idea used for non-deterministic choice agent $\sum$. To model $\sum$, it was enough by determining the first condition than can be deduced and then activate the process associated to it. For probabilistic choice, we need to check the conditions after calculating a *fixpoint*, because we need to know all the conditions that can be entailed before calculating the probabilistic distribution. When multiple probabilistic choice $\bigoplus$ operators are nested, we need to calculate a *fixpoint* for each nested level.

By implementing cells and probabilistic choice it would be easy to implement the model proposed for this work. Valencia proposed in [40] to develop model checking tools for `ntcc`. In the future, we propose using model checking tools for verifying properties of complex systems, such as ours.

In addition, Pérez and Rueda proposed in [20] exploring the automatic generation of models for probabilistic model checker such as *Prism* [12]. The reader should be aware that *Prism* has been used successfully to check properties of real-time systems. We believe that this approach can be used to verify properties in our system.

Finally, we found out that the *time-units* in *Ntccrt* do not represent uniform *time-units*, because in the stand-alone simulation they have different durations. This is a problem when synchronizing a `ntcc` program with other programs. To fix it, we made the duration of each *time-unit* take a fixed time. We did that easily by using the clock provided by Pd or Max and providing a clock input in *Ntccrt* plugins. That way we only start simulating a new *time-unit* once we receive a clock pulse.

On the other hand, fixing the duration of a *time-unit* has two problems. First, if the fixed time is less than the time required to compute all the processes in a *time-unit*, this makes the simulation incoherent. Second, it makes the simulation last longer because the fixed time has to be an upper limit for the *time-unit* duration.

6.5 Related work

Lman was developed as a framework to program RCX Lego Robots. It is composed of three parts: an abstract machine, a compiler and a visual language. We borrowed from this interpreter the idea of having several queues for storing `ntcc`'s processes, instead of using threads. Regrettably, since *Lman* only supports finite domain constraints.

NtccSim was used to simulate biological models [11]. It was developed in Mozart-Oz [26]. It is able to work with finite domains (FD) and a constraint system to reason about real numbers. We conjecture (it has not been proved) that using Mozart-Oz for writing a `ntcc` interpreter it is not as efficient as using *Gecode*, based on the results obtained in the benchmarks of Gecode, where Gecode performs better than Mozart-Oz in constraint solving.

Rueda's sim was developed as a framework to simulate multimedia semantic interaction applications. This interpreter was the first one representing rational trees, finite domain , and finite domain sets constraint systems. One drawback of this interpreter is the use of *Screamer* [34] to represent the constraint systems. *Screamer* is a framework for constraint logic programming written in Common Lisp. Unfortunately, *Screamer* is not designed for high performance. This makes the execution of the `ntcc` models in *Rueda's sim* not suitable for real-time interaction.

7 Tests and Results

Since the creation of *Lman*, performance and correctness have been the main issues to evaluate a `ntcc` interpreter. *Lman* was a great success in the history of `ntcc` interpreters because by using *Lman* it was possible to program Lego^{TM} Robots, and formally predict the behavior of the robots. A few years later, *Rueda's sim* was capable to model multimedia interaction systems.

Although, it is beyond the scope of this research to evaluate whether those interpreters are faster than *Ntccrt* or whether they are able to interact in real-time with a human player, we conjecture that they are not appropriate for real-time interaction for simulating hundreds of *time-units* in complex models such as *Ccfomi*, based on the results presented by their authors.

In this chapter we want evaluate the performance of our `ntcc` interpreter prototypes and also to evaluate the behavior of *Ntccrt*. In order to achieve these goals, we performed different tests to *Ntccrt*

and to our previous implementations of `ntcc`.

First, we tried to develop a generic implementation of lightweight threads that could be used in Lispworks. The purpose was to use threads to manage concurrency in `ntcc` interpreters. We compared Lisp processes (medium-weight threads), our implementation of threads based on continuations, and our implementation of threads based on event-driven programming.

We found out that continuations are not efficient in Lispworks. We also found out that the event-driven implementation of threads is faster than using Lisp processes or continuations. However, it is very difficult to express instructions such as go-to jumps, exceptions and local variable definition on event-driven programming.

Then, we tried using both Lisp processes and the event-driven threads to implement `ntcc` interpreters (explained in Appendix **??**). We found out that context-switch of threads and the fact that it checks for stability constantly adds an overhead in the performance on the `ntcc` interpreter. For those reasons, we discarded using threads for the `ntcc` interpreter. We also found out that encoding `ntcc` processes as Gecode propagators outperforms the threaded implementations of the interpreter.

Each test presented in this chapter was taken with a sample of 100 essays. Time was measured using the *time* command provided by Mac OS X and the *time* macro provided by Common Lisp. All tests were performed under Mac OS X 10.5.2 using an Imac Intel Core 2 duo 2.8 Ghz and Lispworks Professional 5.02.

In the graph bars, we present the average of those samples. The vertical axe is measure in seconds in all graphs. We do not present standard deviation nor other statistical information because the differences of performances between one implementation and another were considerable high to reason about the performance of the implementations. Sometimes, we do not present all the bars in a graph because they do not fit the scale of the graph.

7.1 Testing *Ntccrt* performance

In order to test *Ntccrt* performance, we made two tests. First, we compared a `ntcc` specification to find paths in a graph with other three implementations. Second, we tested *Ccfomi* using *Ntccrt*. Recall from the beginning of this chapter that each test was taken with a sample of 100 essays. Time was measured using the *time* command provided by Mac OS X and the *time* macro provided by Common Lisp. All tests were performed under Mac OS X 10.5.2 using an Imac Intel Core 2 duo 2.8 Ghz and Lispworks Professional 5.02.

7.2 Test: Comparing implementations of `ntcc` interpreters

We compared the execution times of simulating the specification presented to find paths in graph concurrently (explained in detail in Appendix **??**) running on the event-driven Lisp interpreter and *Ntccrt*. We also compared them with a concurrent constraint implementation on Mozart/OZ and a recursive implementation in Lisp (fig. 18).

7.3 Test: Executing *Ccfomi*

Ccfomi is able to receive up to one note each *time-unit*. A reasonable measure of performance is the average duration of a `ntcc` *time-unit* during the simulation. We ran *Ccfomi* in *Ntccrt* with a player interpretating at most the first 300 notes of J.S Bach's two-part Invention No. 5, as studied in [5]. The player chooses (non-deterministically) to play a note or postpone the decision for the next *time-unit*. It took an average of 20 milliseconds per *time-unit*, scheduling around 880 processes per time-unit, and simulating 300 *time-units*. We simulated these experiment 100 times. Detailed results can be found at Appendix **??**. We do not present musical results, since it is out of the scope of this work to conclude whether *Ccfomi* produces or not an improvisation "appealing to the ear". We are only interested on performance tests.

Pachet argues in [19] that an improvisation system able to learn and produce sequences in less than 30ms is appropriate for real-time interaction. Since *Ccfomi* has a response time of 20ms in average for a 300 *time-units* simulation, we conclude that it is capable of real-time interaction according to Pachet's research.

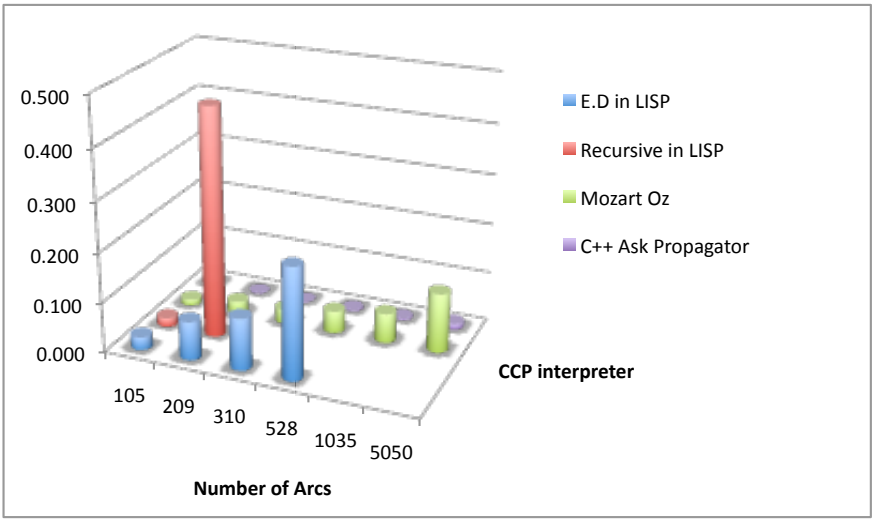


Figure 18: Comparing implementation to find paths in a graph

7.4 Summary

Ntccrt, our `ntcc` interpreter based on encoding `ntcc` processes as Gecode propagators outperforms our threaded and our event-driven implementations of `ntcc`.

Since we are learning and producing sequences with a response time lower than 30 milliseconds then, according to the authors of the *Continuator*, we have a system fast enough to interact with a musician.

7.5 Related work

Lman's developers ran a specification to play a MIDI pitch with a fixed duration each *time-unit* [15]. The tests were made using a Pentium III 930 MHz, 256 MB Ram, Linux Debian Woody (3.0), and the RCX 2.0 Lego robot with running BrickOS 2.6.1. They made a simulation with 100 *time-units*.

This simple process takes an average of 281.25 ms to run each *time-unit* using *Lman*, unfortunately it is not suitable for real-time interaction in music, even if we would run it on modern computers.

On the other hand, Rueda's interpreter ran *Ccfomi* on a 1.67 GHz Apple PowerBook G4 using Digitool's MCL version of Common Lisp, taking an average of 25 milliseconds per *time-unit*, scheduling around 20 concurrent processes. They also made a simulation with 100 *time-units*.

Unfortunately, Rueda's implementation uses some MCL's functions (not defined in the Common Lisp standard) and we were not able to run his interpreter in Mac OS X Intel to compare it with *Ntccrt*. On the other hand, *Lman* is designed for Linux and it is no longer maintained for current versions of Linux and Tcl/tk.

8 Conclusions

In this chapter, we present a summary of the article, some concluding remarks, and we propose some future work thoughts.

8.1 Summary

- We explained how we can model music improvisation using probabilities, extending the notion of non-deterministic choice described in Ccfomi. Although this idea is very simple, the probabilities are computed in constant time and space when the FO is built. We managed to preserve the linear complexity in time and space of the FO on-line construction algorithm.

- Calculating the probability of being on a certain scale makes the model more appropriate for certain music genres, but it requires to calculate multiple constants, which vary according to the genre of tonal music where the user is improvising. For that reason, it is discarded.

- We explained how we can change the intensity of the notes generated in the improvisation. This kind of variation in the intensity is something new for machine improvisation systems as far as we know. We believe that this kind of approach, where simple variations can be made while preserving the style learned from the user and being compatible with real-time implementations, should be a topic of investigation in future work.

- We used cells to represent the variables changing from a *time-unit* to another. Using cells we modeled a probabilistic distribution that changes according to the user and computer interaction. As far as we know, this is the first `pntcc` model where probabilistic distributions change between *time-units*. Unfortunately, current version of *Ntccrt* does not support cells nor probabilistic choice.

- We ran *Ccfomi* in *Ntccrt* taking an average of 20 milliseconds per time-unit (see Chapter 7). Since we are learning and producing sequences with a response time less than 30 milliseconds then, according to the authors of the Continuator, we have a system fast enough to interact with a musician.

- Although Gecode was designed for solving combinatory problems using constraints, we found out that using Gecode for Ntccrt give us outstanding results for writing a `ntcc` interpreter.

- Unfortunately, the interpreter is not able to execute processes leading the *store* to false. However, processes reasoning about a false *store* can be rewritten in a different way, avoiding this kind of situations.

8.2 Concluding remarks

We show how we can make a probabilistic extension of *Ccfomi* using the Factor Oracle. This extension has three main features. First, it preserves the linear time and space complexity of the on-line Factor Oracle algorithm. The Factor Oracle was chosen as the data structure to capture the user style in *Ccfomi* because of its linear complexity. Our extension would not be worth if we had changed the complexity fo the Factor Oracle on-line construction algorithm in order to add probabilistic information to the model, making it incompatible with real-time.

Second, we are using `pntcc` (a probabilistic extension of `ntcc`) for our model. The advantage of `pntcc` is that we do not need to to model all the processes in a new calculus to extend *Ccfomi*, instead we use `pntcc` where we have all the agents defined in `ntcc` (except the $*$ agent, which is not used in this work) and a new agent for probabilistic choice. Adding probabilistic choice to *Ccofmi*, we avoid loops without control during the improvisation that may happen without control in *Ccfomi* due to its non-deterministic nature. In addition, changing the probability distribution, we could favor repetitions in the improvisation, if desired.

Third, the variation in the intensity during the improvisation. This is, as far as we know, the first model considering this kind of variation. Generating variations in the intensity during improvisation, we avoid sharp differences between the user and computer intensity, making the improvisation appealing to the ear (according the musicians we interviewed). Variations in the musical attributes are well-known for decades in Computer Assisted Composition, but in interactive systems (such as machine improvisation) variations are still an open subject, in part, due to the real-time requirements of the interactive systems.

If the reader does not consider relevant using process calculi (such as `pntcc`) to model, verify and execute a real-time music improvisation system, we pose the reader the following questions. Has the reader developed a real-time improvisation system on a programming language mixing non-deterministic and probabilistic choices? Try verifying the system formally! Is it an easy task? Would the reader be able to write such system in 50 lines of code? Using `pntcc`, we did it.

If we can model such systems using `pntcc` and process calculi have been well-known in theory of concurrency for the past two decades, why they have not been used in real-life applications? Garavel argues that models based on process calculi are not widespread because there are many calculi and many

variants for each calculus, being difficult to choose the most appropriate. In addition, it is difficult to express an explicit notion of time and real-time requirements in process calculi. Finally, he argues that existing tools for process calculi are not user-friendly.

We want to make process calculi widespread for real-life applications. We left the task of representing real-time in process calculi and choosing the appropriate variant of each calculus for each application to senior researchers. This work focuses on developing a real-life application with `pntcc` and showing that our interpreter *Ntccrt* is a user-friendly tool, providing a graphical interface to describe `ntcc` processes easily and compile models such as *Ccfomi* to efficient C++ programs capable of real-time user interaction. We also showed that our approach to design *Ntccrt* offers better performance than using threads or event-driven programming to represent the processes.

The reader may argue that although we can synchronize *Ntccrt* with an external clock provided by Max or Pd, this does not solve the problem of simulating models when the clock step is smaller than the time necessary to compute a *time-unit*. In addition, the reader may argue that we encourage formal verification of `ntcc` and `pntcc` models, but there is not an existing tool to verify these models automatically, not even semi-automatically.

The reader is right! For that reason, currently the Avispa research group (sponsored by Pontificia Universidad Javeriana de Cali) is developing an interpreter for an extension of `ntcc` capable of modeling *time-units* with fixed duration. In addition, Avispa is proposing to Colciencias a project called Robust theories for Emerging Applications in Concurrency Theory: Processes and Logic Used in Emergent Systems (REACT-PLUS). REACT-PLUS will focus on developing verification tools for `ntcc`, `pntcc` and other process calculi. In addition, the project will continue developing faster and easier to use interpreters for them.

We invite the reader to read the following section to know about the future work thoughts that we propose. In addition, the reader may find more information about the REACT-PLUS proposal at http://www.lix.polytechnique.fr/comete/pp.html.

8.3 Future work

In the future, we propose extending our research in the following directions.

8.4 Extending our model

We propose capturing new elements in the music sequences. For instance, considering the music timbre, music pitch/amplitude variation (e.g., vibrato, bending and acciaccatura), and resonance effects (e.g., delay, reverb and chorus).

8.5 Improvisation set-ups

Several concurrent improvisation situation set-ups have been proposed [3], [6], but none of them have been implemented for real-time music improvisation. Rueda et al. in [28] propose the following set-ups: n performers and n oracles learning and performing; one performer, one oracle learning, and several improvisation processes running concurrently in the same oracle; one performer and several oracles learning from different viewpoints of the same performance.

8.6 Using Gelisp for *Ntccrt*

Currently, we are only using Lisp to generate C++ code. However, it is not possible to embed Lisp code in the interpreter. To work around that, we propose using *Gelisp* for writing a new interpreter, taking advantage of the call-back functions provided by the Foreign Function Interface (FFI) to call Lisp functions from C++. That way a process can trigger the execution of a Lisp function.

8.7 Adding support for cells for *Ntccrt*

The idea for a real-time capable implementation of cells is to extend the implementation of persistent assignation. Cells, in the same way than persistent assignation, require to pass the domain of a variable

from the current *time-unit* to a future *time-unit*.

8.8 Developing an interpreter for `pntcc`

Pérez and Rueda already propose an interpreter for `pntcc`. To achieve probabilistic choice in Ntccrt, we propose extending the idea used for non-deterministic choice agent $\sum$. To model $\sum$, it was enough by determining the first condition that can be deduced and then activate the process associated to it. For probabilistic choice, we need to check the conditions after calculating a *fixpoint*, because we need to know all the conditions that can be entailed before calculating the probabilistic distribution. When multiple probabilistic choice $\bigoplus$ operators are nested, we need to calculate a *fixpoint* for each nested level.

8.9 Developing an interpreter for `rtcc`

There is not a way to describe the behavior of a `ntcc` *time-unit* if the fixed time is less than the time required to execute all the processes scheduled. For that reason, we propose developing an interpreter for the *Real Time Concurrent Constraint* (`rtcc`) [32] calculus.

`Rtcc` is an extension of `ntcc` capable of dealing with strong time-outs. Strong time-outs allow the execution of a process to be interrupted in the exact instant in which internal transitions cause a constraint to be inferred from the *store*. `Rtcc` is also capable of delays inside a single time unit. Delays inside a single time unit allows to express things like "this process must start 3 seconds after another starts". Sarria proposed in [32] developing an interpreter for `rtcc`. We believe that we can extend *Ntccrt* to simulate `rtcc` models.

8.10 Adding other graphical interfaces for *Ntccrt*

For this work, we conducted all the tests under Mac OS X using Pd and stand-alone programs. Since we are using Gecode and Flext to generate the externals, they could be easily compiled to other platforms and for Max. We used Openmusic to define an iconic representation of `ntcc` models. In the future, we also propose exploring a way of making graphical specifications for `ntcc` similar to the graphical representation of data structures in Pd.

8.11 Developing model checking tools for *Ntccrt*

Valencia proposed using model checking tools for verifying properties in complex `ntcc` models. In addition, Pérez and Rueda proposed developing model checking tools for `pntcc`. For instance, they propose exploring the automatic generation of models for *Prism* based on a `pntcc` model. We propose generating models to existing model checkers automatically to prove properties of the systems before simulating them on *Ntccrt*.

Acknowledgements

To Carlos Olarte from Lix, Carlos Agón and Jean Bresson from Ircam, Antal Buss from Pujc, Guido Tack from Saarland university, Christian Schulte from KTH, Luis Omar Quesada from 4C, Boris Mejias and Gustavo Gutiérrez from UCL for their valuable comments and their help while I was developing the real-time interpreter. Specially, to Gustavo for his enormous patience.

For the development of the model, I want to thank Frank Valencia and Catuscia Palamidessi from Lix, Jorge Pérez from university of Bologna, and Gérard Assayag and Arshia Cont from Ircam. To my music teachers, who taught me all I know about music and gave me helpful comments in the beginning of this research: Mauricio Santamaría, Alberto Riascos, and Juan Manuel Collazos.

References

[1] Cyril Allauzen, Maxime Crochemore, and Mathieu 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] Gérard Assayag and Georges Bloch. Navegating the oracle: A heuristic approach. In *International Computer Music Conference '07*, pages 405–412, Copenhagen, Denmark, August 2007.

[3] Gérard Assayag and Shlomo Dubnov. Improvisation planning and jam session design using concepts of sequence variation and flow experience. In *Sound and Music Computing 2005*, Salerno, Italie, Novembre 2005.

[4] Darrell Conklin and Ian H. Witten. Multiple viewpoint systems for music prediction. *Journal of New Music Research*, 24:51–73, 1995.

[5] Arshia Cont. Antescofo: Anticipatory synchronization and control of interactive parameters in computer music. In *Proceedings of International Computer Music Conference (ICMC)*. Belfast, August 2008.

[6] Arshia Cont, Shlomo Dubnov, and Gérard Assayag. A framework for anticipatory machine improvisation and style imitation. In *Anticipatory Behavior in Adaptive Learning Systems (ABiALS)*, Rome, Italy, Septembre 2006.

[7] Arshia Cont, Shlomo Dubnov, and G.Assayag. Anticipatory model of musical style imitation using collaborative and competitive reinforcement learning. In Butz M.V., Sigaud O., Pezzulo G., and Baldassarre G., editors, *Anticipatory Behavior in Adaptive Learning Systems*, volume 4520 of *Lecture Notes in Computer Science / Artificial Intelligence (LNAI)*, pages 285–306. Springer Verlag, Berlin, 2007.

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

[9] Hubert Garavel. Reflections on the future of concurrency theory in general and process calculi in particular. *Electron. Notes Theor. Comput. Sci.*, 209:149–164, 2008.

[10] G.Assayag and Sholomo Dubnov. Using factor oracles for machine improvisation. *Soft Comput.*, 8(9):604–610, 2004.

[11] Julian Gutiérrez, Jorge A. Pérez, Camilo Rueda, and Frank D. Valencia. Timed concurrent constraint programming for analyzing biological systems. *Electron. Notes Theor. Comput. Sci.*, 171(2):117–137, 2007.

[12] 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.

[13] A. Lefebvre and T. Lecroq. Computing repeated factors with a factor oracle. In L. Brankovic and J. Ryan, editors, *Proceedings of the 11th Australasian Workshop On Combinatorial Algorithms*, pages 145–158, Hunter Valley, Australia, 2000.

[14] Eduardo Reck Miranda. *A-Life for Music: Music and Computer Models of Living Systems (Computer Music and Digital Audio Series)*. A-R Editions, 2011.

[15] M.P. Muñoz and A.R. Hurtado. Lman ntcc: Abstract machine for concurrent programming of lego robots. (in spanish). B.Sc. Thesis, Department of Computer Science and Engineering, Pontifica Universidad Javeriana, Cali, 2004.

[16] Pilar Muñoz and Andrés Hurtado. Programming robot devices with a timed concurrent constraint programming. In *In Principles and Practice of Constraint Programming - CP2004. LNCS 3258, page 803.Springer*, 2004.

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

[18] C. Olarte and C. Rueda. A stochastic non-deterministic temporal concurrent constraint calculus. *sccc*, 0:30–36, 2005.

[19] François Pachet. Playing with virtual musicians: the continuator in practice. *IEEE Multimedia*, 9:77–82, 2002.

[20] Jorge Pérez and Camilo Rueda. Non-determinism and probabilities in timed concurrent constraint programming. In Maria Garcia de la Banda and Enrico Pontelli, editors, *Logic Programming, 24th Internatinoal Conference, ICLP 2008, Udine, Italy, December 9-13 2008, Proceedings*, volume 5366 of *Lecture Notes in Computer Science*, pages 677–681, pntcc probabilities non-determinism 2008.

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

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

[23] 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.

[24] Viswanath Ramachandran and Pascal Van 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.

[25] Andreas Rossberg, Guido Tack, and Leif Kornstaedt. Status report: Hot pickles, and how to serve them. In Claudio Russo and Derek Dreyer, editors, *2007 ACM SIGPLAN Workshop on ML*, pages 25–36. ACM, 2007.

[26] Peter Van Roy, editor. *Multiparadigm Programming in Mozart/Oz, Second International Conference, MOZ 2004, Charleroi, Belgium, October 7-8, 2004, Revised Selected and Invited Papers*, volume 3389 of *Lecture Notes in Computer Science*. Springer, 2005.

[27] C. Rueda and F. Valencia. On validity in modelization of musical problems by ccp. *Soft Comput.*, 8(9):641–648, 2004.

[28] Camilo Rueda, Gérard Assayag, and Shlomo Dubnov. A concurrent constraints factor oracle model for music improvisation. In *XXXII Conferencia Latinoamericana de Informtica CLEI 2006*, Santiago, Chile, Aot 2006.

[29] Camilo Rueda and Frank Valencia. A temporal concurrent constraint calculus as an audio processing framework. In *SMC 05*, 2005.

[30] Camilo Rueda and Frank D. Valencia. Formalizing timed musical processes with a temporal concurrent constraint programming calculus. In *Proc. of Musical Constraints Workshop CP2001*, 2002.

[31] Vijay A. Saraswat. *Concurrent Constraint Programming.* MIT Press, 1992.

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

[33] Christian Schulte. Programming deep concurrent constraint combinators. In *Practical Aspects of Declarative Languages, Second International Workshop, PADL 2000, volume 1753 of Lecture Notes in Computer Science*, pages 215–229. Springer-Verlag, 2000.

[34] Jeffrey Mark Siskind and David Allen McAllester. Nondeterministic lisp as a substrate for constraint logic programming. In *Proceedings of the 11th National Conference on Artificial Intelligence*, July 1993.

[35] Mauricio Toro, Rueda Camilo, Agón Carlos, and Assayag Gérard. Ntccrt: A concurrent constraint framework for soft real-time music interaction. *Journal of Theoretical & Applied Information Technology*, 82(1), 2015.

[36] Mauricio Toro, Myriam Desainte-Catherine, and Julien Castet. An extension of interactive scores for multimedia scenarios with temporal relations for micro and macro controls. *European Journal of Scientific Research*, 137(4):396–409, 2016.

[37] Mauricio Toro, Myriam Desainte-Catherine, and Camilo 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] Mauricio Toro, Anna Philippou, Sair Arboleda, María Puerta, and Carlos M. Vélez S. Mean-field semantics for a process calculus for spatially-explicit ecological models. In César A. Muñoz and Jorge A. Pérez, editors, Proceedings of the Eleventh International Workshop on *Developments in Computational Models,* Cali, Colombia, October 28, 2015, volume 204 of *Electronic Proceedings in Theoretical Computer Science*, pages 79–94. Open Publishing Association, 2016.

[39] Mauricio Toro, Camilo Rueda, Carlos Agón, and Gérard Assayag. Gelisp: A framework to represent musical constraint satisfaction problems and search strategies. *Journal of Theoretical & Applied Information Technology*, 86(2), 2016.

[40] Frank D. Valencia. Decidability of infinite-state timed ccp processes and first-order ltl. *Theor. Comput. Sci.*, 330(3):557–607, 2005.