A Concurrent Constraints Factor Oracle Model for Music Improvisation

Camilo Rueda, Gérard Assayag, Shlomo Dubnov

A Concurrent Constraints Factor Oracle Model for Music Improvisation

Camilo Rueda[1], Gérard Assayag[2], and Shlomo Dubnov[3]

[1] Universidad Javeriana-Cali, Department of Science and Engineering of Computing
crueda@cic.puj.edu.co
[2] IRCAM, Music Representations Team, Paris, assayag@ircam.fr
[3] University of California, San Diego, sdubnov@ucsd.edu

Summary. Machine improvisation and related style simulation problems usually consider building representations of time-based media data, such as music, either by explicit coding of rules or applying machine learning methods. *Stylistic learning* applies such methods to musical sequences in order to capture salient musical features and organize these features into a model. The *Stylistic simulation* process browses the model in order to generate variant musical sequences that are stylistically consistent with the learned material. If both the learning process and the simulation process happen in real-time, in an interactive system where the computer "plays" with musicians, then Machine Improvisation is achieved. Improvisation models have to cope with a trade-off between completeness (all the possible patterns and their continuation laws are discovered) and incrementality (the completeness is ensured only asymptotically for infinite sequences). In a previous work we devised a complete and incremental model based on the Factor Oracle Algorithm. In this paper we propose a concurrent constraints model for the Factor Oracle and show how it can be used in a concurrent learning/improvisation situation. Our model is based on a non-deterministic concurrent constraint process calculus (NTCC). Such an approach allows the system to respond in a faster and more flexible manner to real-life performance situations. In addition, the declarative nature of constraints greatly simplifies the expansion of the system with improvisation rules at a higher musical level. We also describe the implementation of our model in a NTCC interpreter written in Common Lisp that is capable of real time performance.

Key words: Factor oracle, concurrent constraints process calculus, constraint programming, machine learning, machine improvisation

1 Introduction

Machine improvisation and related style simulation problems usually consider building representations of time-based media data, such as music, either by explicit coding of rules or applying machine learning methods. We call *stylistic learning* the process of applying such methods to musical sequences in order to capture salient musical features and organize these features into a model. The *Stylistic simulation* process browses the model in order to generate variant musical sequences that are stylistically consistent with the learned material. While it is hard to validate this consistency, which involves some value judgement, it may however be sustained by two types of experiments : successful style classification using the same model [3], and experimental psychology protocols in the form of an extended"Turing test" [6]. If both the learning process and the simulation process

happen in real-time, in an interactive system where the computer "plays" with musicians, then Machine Improvisation is achieved.

The learning schemes we have investigated so far belong to the family of statistical modeling, more specifically to *context-inference* modeling. This kind of modeling as applied to musical sequences has been experimented since the very beginnings of computer music [Con03]. The idea behind is that events in a musical piece can be predicted from the sequence of preceding events. The operational property of such models is to provide the conditional probability distribution over an alphabet given a preceding sequence called a context. This distribution is used for predicting the next symbol providing the context already generated.

First experiments in context based modeling made intensive use of Markov chains, based on an information-theoretic principle : complex sequences do not have obvious underlying source, however, they may exhibit a so-called *short memory property* [8] : there exists a certain memory lengh L such that the conditional probability distribution on the next symbol does not change significantly if we condition it on *contexts* longer than L. This justifies the markovian approaches traditionally used in computer music.

These models have to cope with a trade-off between completeness (all the possible patterns and their continuation laws are discovered) and incrementality (the completeness is ensured only asymptotically for infinite sequences). In order to reduce the model complexity, Variable Memory Markov models (VMM) have successfully replaced previous fixed order markov automata.

In the music domain, we have proposed in earlier works a method for building musical style analyzers and generators based on several algorithms for prediction of discrete sequences using VMM. The class of these algorithms is large and we focused mainly on two variants of predictors - universal prediction based on Incremental Parsing (IP) and prediction based on Probabilistic Suffix Trees (PST) [4] [7]. These methods were either incomplete or not incremental. More recently, we have introduced a new method based on the Factor Oracle Algorithm, which is complete and incremental [1]. This method has been implemented in a purely sequential way in the OpenMusic environment and succesfully tested in real-life improvisation situations (Omax system). Due to its sequential nature, Omax cannot learn and improvize at the same time.

In this paper we propose to go one step further by providing a Concurrent Constraints model for the factor oracle, showing how it can be used in a concurrent learning/improvisation situation. The benefits of this new approach are numerous : the system will be more interactive ; it will respond in faster and more flexible manner to real-life performance situations ; due to the declarative nature of constraints, it wil lbe easy to expand by adding rules at a higher musical level ; within the power limitations of the machine, as many concurrent oracles will be able to work silmutaneously, in order to model either multi-voice improvisation or multiple musical viewpoints ; it will be able to learn on several musicians simultaneously.

2 Background

Our model is based on a concurrent constraint implementation of factor oracles. In this section we describe both.

2.1 Factor Oracle

We give here a short description of the properties and construction of factor oracles. The formal definitions can be found in [9]. Basically a factor oracle is a finite state automaton constructed in

linear time and space in an incremental fashion. A sequence of symbols $s = \sigma_1\sigma_2 \dots \sigma_n$ is learned in such an automaton, which states are $0, 1, 2 \dots n$. There is always a transition arrow (called factor link) labelled by symbol σ_i going from state $i-1$ to state $i, 1 \leq i < n$. Depending on the structure of s, other arrows will be added to the automaton. Some are directed from a state i to a state j, where $0 \leq i < j \leq n$. These also belong to the set of factor links and are labelled by symbol σ_j. Some are directed "backwards", going from a state i to a state j, where $0 \leq j < i \leq n$. They are called suffix links, and bear no label. The factor links model a factor automaton, that is every factor p in s corresponds to a unique factor link path labeled by p, starting in 0 and ending in some other state. Suffix links have an important property : a suffix link goes from i to j iff the longest repeated suffix of $s[1..i]$ is recognized in j. Thus suffix links connect repeated patterns of s.

The oracle is learned on-line (see figure 1). For each new entering symbol σ_i, a new state i is added and an arrow from $i-1$ to i is created with abel 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 originates (going to some state j), or until there is 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 the state j or to state 0 depending on which condition terminated the iteration. Navigating the

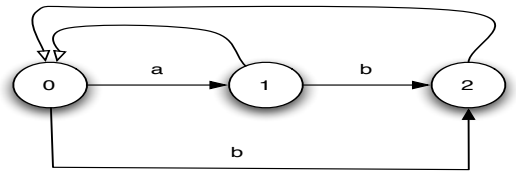

Fig. 1. A FO automaton for $s = abb$

oracle in order to generate variants is straightforward : starting in any place, following factor links generates a sequence of labelling symbols that are repetitions of portions of the learned sequence; following one suffix link followed by a factor links creates a recombined pattern sharing a common suffix with an existing pattern in the original sequence. This common suffix is, in effect, equivalent to the context in the context-inferences model. In addition to completeness and incrementality of this model, the best suffix is known at the minimal cost of just following one pointer. By following more than one suffix link before going back to the factor generation, or by reducing the number of successive factor link steps, we make the generated variant less resemblant to the original.

2.2 The Non deterministic Temporal Concurrent Constraint Calculus

Concurrent constraint programming (CCP [13]) is intended as a model of concurrent systems. In CCP a concurrent system is modeled in terms of constraints over the variables of the system. A constraint is a formula representing *partial information* about the values of some of the variables. For example, in a system with variables $pitch_1, pitch_2$ taking MIDI values, the constraint $pitch_1 > pitch_2 + 2$ specifies possible values for $pitch_1$ and $pitch_2$ (those where $pitch_1$ is at least a tone higher than $pitch_2$). The CCP model includes a set of (*basic*) constraints and a so-called *entailment relation* $\models$ between constraints. This relation gives a way of *deducing* a constraint from the information supplied by other constraints. For example, $pitch_1 > pitch_2 + 2,\ pitch_2 > 60 \models pitch_1 > 48$.

Computation in the CCP model proceeds by accumulating information (i.e. constraints) in a *store.* The information specifies all that is known about the values of the variables at a given moment. Information on the *store* may increase but it cannot decrease. Concurrent processes interact with the store either *telling* new information or *asking* whether some constraint can be deduced (entailed) from the information contained in it. It may well happens that the constraint cannot be entailed. In this case the interacting process is said to *block* until some other processes tell enough information to the store to deduce its constraint.

One drawback of the CCP model as presented above is that information is always accumulated. There is no way to eliminate it. This poses difficulties for modeling reactive systems in which information on a given variable changes depending on the interactions of a system with its environment, as is the case, for example, in interactive musical improvisation systems. Different extensions on the CCP model have been proposed to handle reactive systems. One such model is the non-deterministic temporal concurrent constraint calculus (NTCC, [10]). This calculus introduces the notion of time, seen as a sequence of *time units*. At each time unit a CCP computation takes place, starting with an empty store (or one that has been given some information by the environment). Concurrent constraints agents operate on this store as in the usual CCP model to accumulate information into the store. As opposed to the CCP model, however, the agents can schedule processes to be run in future temporal units. In addition, since at the beginning of each time unit a new store is created information on the value of a variable can change (e.g. it can be forgotten) from one unit to the next. The computational agents of NTCC are describe in table 1. Intuitively, agent **tell**(c) adds

Agent	meaning
tell(c)	Add c to the current store
when c **do** A	if c holds now, run A
local x **in** P	run P with local x
$A \parallel B$	Parallel composition
next A	run A at the next instant
unless c **next** A	unless c can be inferred now, run A
$\sum_{i \in I}$ **when** c_i **do** P_i	choose P_i s.t. c_i holds
$*\, P$	delay P undefinitely (not forever)
$!\, P$	Execute P each time unit (from now)

Table 1. NTCC agents

information c to the store of the current time unit. This information can then be used to deduce other constraints. Agent **when** c **do** A asks whether c can be deduced to hold from the current store and if so, executes agent A. Computed information that is to remain local to an agent is defined by **local** x **in** P. Here, information on x added by P is only seen by itself or by its subprocesses (if any). Reciprocally, any existing global information on x cannot be seen by P. The parallel composition agent $A \parallel B$ runs A and B in parallel. Agent **next** A schedules A to be run at the next time unit. Notice that an agent **next tell**(c) adds information c to the store of the next time unit. Notice that this store might initially be empty or contain some information provided externally by the environment (e.g. as the result of the system interacting with a musician), but is completely independent of the store of the current time unit. Agent **unless** c **next** A offers the possibility of performing activity on the basis of *absence* of information. When constraint c cannot be deduced from the store of the current time unit, action A is performed in the next time unit.

It should be noticed that in NTCC this means that entailment checking of c is performed when all other processes have finished, i.e. when it is certain that c cannot be deduced in the current time unit.

The choice agent $\sum_{i \in I}$ **when** c_i **do** P_i *non deterministically* runs some process P_i such that its guard c_i can be deduced from the current store. Several of the c_i's could hold but only one P_i is non deterministically chosen. Agent $*P$ schedules P to be run either now or at some unspecified time in the future. In NTCC, agents are ephemeral. Their life span is just the time unit in which they run. Agent $!P$ adds persistence. It launches process P at the current time unit and at all future time units.

The following example illustrates computation in NTCC.

$$SYST \stackrel{\text{def}}{=} \ !\,\textbf{tell}(start > 20) \ \parallel \ CHECK \ \parallel \ PLAY \ \parallel \ *\ \textbf{tell}(play(done)) \ \parallel \ BEAT(0)$$

$$PLAY \stackrel{\text{def}}{=} \ !\,\sum_{i \in \{1,2,3\}} \textbf{when}\ play(on)\ \textbf{do}\ NOTE_i$$
$$CHECK \stackrel{\text{def}}{=} \ \textbf{unless}\ beat < start\ \textbf{next}\, play(on) \ \parallel \ \textbf{unless}\ play(done)\ \textbf{next}\, CHECK$$
$$BEAT(i) \stackrel{\text{def}}{=} \ \textbf{tell}(beat = i) \ \parallel \ \textbf{next}\, BEAT(i+1)$$

The system asserts (persistently) that the value of *start* is greater than 20 and runs in parallel three processes $PLAY$, $CHECK$ and $BEAT$. It also launches a process that is to stop performance at some unspecified time unit in the future. Process $PLAY$ non deterministically chooses one of three notes when playing is *on*. Process $CHECK$ asserts that playing is *on* once it can be deduced that the beat counter is greater than or equal to the starting time. It does so repeatedly until the stop playing signal arrives. The $BEAT$ process is simply a counter (recursive process definitions can be encoded in the standard NTCC calculus. See [10]).

The NTCC calculus has an associated linear temporal logic. Desirable properties of an NTCC model can be expressed as a formula in this logic. A proof system allows then to verify wether the NTCC model satisfies or not the property.

The NTCC calculus has been used to model synchronization of concurrent musical processes ([12]). We use it here to account for the concurrent interaction of Factor Oracle learning and improvisation processes with a musician during performance.

3 The NTCC model for FO improvisation

The Factor Oracle automaton generation process is shown below. The system consists of three subsystems, learning (ADD), improvisation ($CHOICE$) and playing ($PLAYER$) running concurrently. A synchronization process ($SYNC$) decides when a new symbol can be learned. The system uses three kinds of variables to represent the partially built Factor Oracle automaton. Variables $from_k$ denote the set of labels of all currently existing factor links going forward from k. Variables S_i are suffix (i.e. backward) links from each state i and variables $\delta_{i,\sigma}$ give the state reached from i by following a factor link labeled σ. The automaton of figure 1, for example, can be represented by $from_0 = \{a, b\}, from_1 = \{b\}, S_1 = 0, S_2 = 0, \delta_{0,a} = 1, \delta_{0,b} = 2, \delta_{1,b} = 2$.

The ntcc processes below incrementally extend an automaton by adding information on those variables. Process $LOOP_i(k)$ adds (if needed) factor links labeled σ_i to state i from all states k reached from $i-1$ by backward links, then computes S_i, the suffix link from i.

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

The process adding state i and working backwards through the automaton is the following:

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

The two processes above model the learning phase. The learning and improvisation phases can be done concurrently. They must proceed in such a way that improvisation always works on a completely built subgraph. This is easily accomplished by synchronizing on S_i. Indeed, when S_i is determined the subgraph up to state i has been completely built. The $SYNC$ process below keeps adding symbols to the automaton provided the previous one has already been added (S_{i-1} is determined) and the performer has already played beyond the currently known symbols ($go \geq i$). Notice that synchronization is greatly simplified by the use of constraints. If at a given moment variable S_i has no value, **when** processes depending on it are blocked.

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

A musician is modeled as a process playing some note p every once in a while. The following process non deterministically choses between playing now or postponing decision to the next time unit. When playing is decided a further non deterministic choice is performed to select some symbol p (the note) from the alphabet. This represents the act of playing.

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

In the above, notation $A + B$, the non-deterministic choice between A and B, is a shorthand for $\sum_{i \in \{1,2\}} \textbf{when } true \textbf{ do } (\textbf{when } i = 1 \textbf{ do } A\ \parallel\ \textbf{when } i = 2 \textbf{ do } B)$.

The improvisation process uses a probability distribution function $\Phi : \mathcal{R} \rightarrow \{0, 1\}$. The process starts from state k and chooses stochastically according to probability q whether to output symbol σ_{k+1} or to follow a backward link S_k and then output some (non deterministically chosen) symbol $\sigma \in from_{S_k}$. When $S_k = -1$ there is no other choice but to output symbol σ_{k+1}. This is modeled as follows:

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

Stochastic constructs are so common in system modeling that NTCC was extended to include them (see [11]). The extension allows prefixing a process with a probability. Process ${}_{\rho}P$ is launched with probability ρ. Using this extension the $CHOICE$ process could be easily extended to do a probabilistic rather than a non deterministic choice of an element $\sigma \in from_{S_k}$.

The whole system is represented by a process doing the appropriate variable initializations and launching all processes. Improvisation is scheduled to start after n symbols have been produced by the player.

$$
\begin{array}{l}
System_{n,p} \stackrel{\text{def}}{=} \\
\quad \parallel \ ! \ \textbf{tell}(q = p) \ \parallel \ ! \ \textbf{tell}(S_0 = -1) \ \parallel \ PLAYER_1 \ \parallel \ SYNC_1 \\
\quad \parallel \ ! \ \textbf{when} \ go = n \ \textbf{do} \ CHOICE_{\Phi}(n)
\end{array}
$$

Initially, information on most variables is missing. Most **unless** processes will then be launched to add information representing the partially constructed automaton. For instance, in the partially constructed automaton of figure 2 a backward link from state 4 to state 3 is missing.

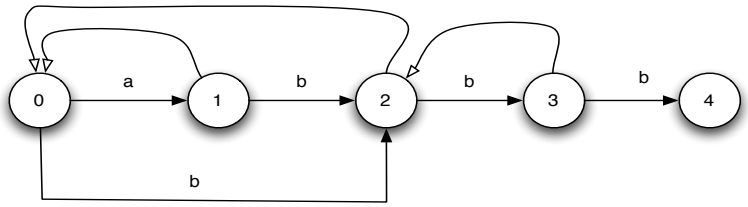

Fig. 2. Partial automaton

In this situation, the above processes will have computed the following (permanent) information:

$$
\begin{array}{l}
\delta_{0,a} = 1, \delta_{0,b} = 2, \delta_{1,b} = 2, \delta_{2,b} = 3, \delta_{3,b} = 4 \\
a \in from_0, b \in from_0, b \in from_1, b \in from_2, b \in from_3 \\
S_0 = -1, S_1 = 0, S_2 = 0, S_3 = 2 \\
\sigma_1 = a, \sigma_2 = b, \sigma_3 = b, \sigma_4 = b
\end{array}
$$

Since the learning process has not yet computed any information on S_4, process $SYNC_5$ is waiting and process $CHOICE_{\Phi}(3)$ (if running) is improvising based on the complete sub-automaton up to state 3.

4 Running the NTCC model

For testing the NTCC model described above we implemented in the Common Lisp programming language a NTCC interpreter. Lisp was used because we plan to integrate the factor oracle improvisation model into the Open Music [14] composition environment. This environment, written in Lisp, contains a great number of music composition tools developed from the expertise of many composers.

In the interpreter each NTCC agent is represented by a concurrent Lisp process. Each process has a particular waiting function. For example, the waiting function of the process implementing a **when** $x > y$ **do tell**$(z = x + y)$ waits until its guard can be deduced to be true. That of a **unless** $x < y + z$ **next tell**$(w = y + z)$ waits until the current time unit is done. The waiting function is used by the Lisp scheduler to decide whether to activate or to defer the process. The architecture of the interpreter is shown in figure 3. First, all user defined NTCC processes are launched causing Lisp to place them in a process queue. The Lisp scheduler repeatedly selects from this queue some enabled process to be run. Each NTCC process is thus run when its waiting function allows (otherwise it is rescheduled).

A concurrent $TICK$ process is permanently testing *stability* of the current time unit. Time unit stability is achieved either when no processes are left in the queue or when all processes have been stopped by their waiting function. When this is the case, the $TICK$ process deletes from the queue all but the *next* and *unless* processes. The *unless* processes are then run and any resulting processes (together with the previously existing *next* processes) are rescheduled for the next time unit.

Each of the mentioned processes run concurrently in a separate thread, following closely the concurrent nature of NTCC.

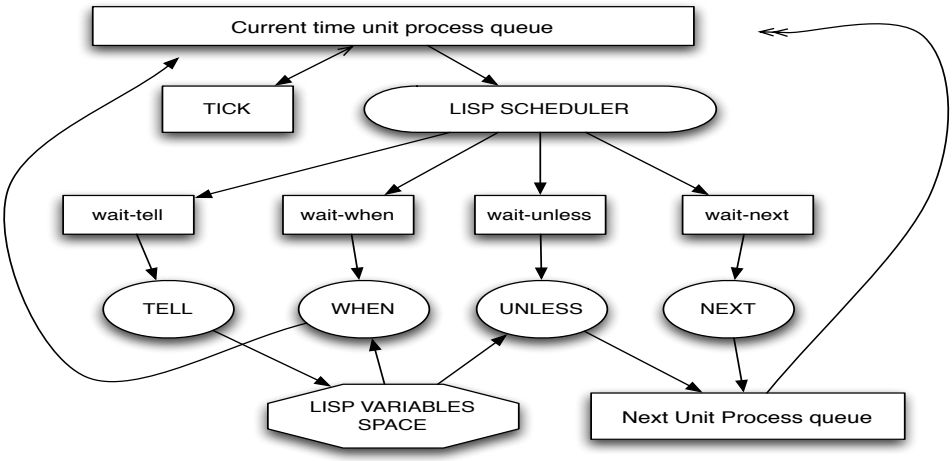

Fig. 3. NTCC interpreter architecture

The Lisp implementation of each NTCC process is similar. A *when* process, for example, is implemented as follows:

```
(defun iswhen (test body vars)
       (if vars (wait-entailed test))
       (eval body))

(defun whenp (test body &rest vars)
       (process-run-function "WHEN" (function iswhen) test body vars))
```

The Lisp primitive "process-run-function" launches a process whose code is given by the "iswhen" function. This code defines "wait-entailed" as waiting function (testing entailment of the *when* guard) and runs the NTCC process comprising the body of *when*. This only occurs if the waiting function ever returns *true*.

In the Factor Oracle model the number of variables is not known a priori since learning is incremental. This usually precludes the use of constraint satisfaction schemes. Being able to handle models with an unknown number of variables is a powerful feature of NTCC. The interpreter easily implements this using the power of Lisp for dynamically translating any string into a variable. For example, a variable $S_{3,5}$ can be constructed and used in the interpreter with

```
(make-variable "S" 3  5 )
(whenp (>  k 0) (tellp  (=  S_3_5 z) ) )
```

Moreover, the user is also given the possibility of launching explicitly a time unit computation. Launching 100 time units is done by

```
(tick-current-time 100)
```

The syntax of NTCC processes in the interpreter closely resembles the corresponding calculus definitions (using Lisp parenthesized prefix notation). For example, process $SYNC_i$ is implemented as

```
(define-process synci (i)
  (let ( (S_i  (make-variable "S" i)))
   (parp (whenp (and (>=  S_i-1 -1) (>= go i))
                (parp  (callp addi i)
                       (nextp (callp synci (1+ i))) )
         (unlessp (and (>=  S_i-1 -1) (>=  go i) )
                  (callp synci i)))))
```

where *parp* stands for the NTCC parallel construct. Recursive NTCC process definitions are thus simply implemented as lisp functions invoked using a special *callp* primitive. The recursive call only takes effect when the enclosing *next* process is executed in the next time unit. The following code concurrently launches the learning process constructing the automaton (function *synci*), the player, the factor oracle improvisation implementing $CHOICE_{\Phi}(k)$ (function *improvise*) and a replicated process continuously setting a note (in MIDI cents notation) when it is available.

```
(parp (callp player 1)
      (callp synci 1)
      (callp improvise 0 40)
      (rep (whenp (>= out 0)
           (tellp (isequal current-note (* out 100))))))
```

The second argument of *improvise* is the probability value in percentage. The above runs concurrently with a further independent process playing every thirtieth of a second a note computed by the factor oracle. If the factor oracle has not yet computed its next note, the previous one is repeated:

```
(defun do-play-tick ()
  (process-wait-with-timeout "time" 2  )
  (play-midi-note current-note 70 300))
```

```
    (do-play-tick )))

(defun play-each-note ( )
  (process-run-function "PLAYING" (function do-play-tick) ) )
```

In the above, value 2 in "process-wait-with-timeout" denotes a 2 sixtieth of a second wait. Parameters 70 and 300 denote fixed volume and duration (in milliseconds) of the played note.

We ran the above system in a 1.67 GHz Apple PowerBook G4 using Digitool's MCL version of Common Lisp. Each NTCC time unit computation took an average of 25 milliseconds. This is fast enough for real time interaction in an actual performance situation. Each time unit schedules around 20 concurrent processes. The indicated time figures are for a system running concurrently the learning and improvisation phases (plus a process simulating the player). In previously reported improvisation systems the used scheme is an iteration of the learning and improvisation phases run in sequence. All synchronization in this concurrent version is transparently ensured by the blocking nature of *when* NTCC processes.

5 Conclusions and future work

We have shown that a concurrent constraints model of a Factor Oracle music improvisation process allows a simple expression of all synchronizing that goes on when both the learning and improvisation phases are done concurrently. The reactive nature of the NTCC calculus is used to simulate in a natural manner the interaction between the system and a musician during performance. We described an implementation of a NTCC interpreter in Common Lisp. The interpreter runs concurrently the learning and improvisation phases in real time. The results presented here are encouraging to develop the model in several directions:

- Improvisation situation set-ups :
  A system comprising
  - n performers and n oracles learning and performing
    The challenge here is maintaining real time performance. Since the additional processes are independent, this should pose no difficulties for modeling.
  - 1 performer, one oracle learning, several improvisation processes running concurrently on the same oracle
  - 1 performer, several oracles learning different viewpoints of the same performance (e.g. pitch, duration, intensity). These oracles have to be put to work together in order to rebuild complete musical data.
- Technical issues :
  in the case of a complex improvisation set-up such as those mentioned, complex synchronization issues arise. When several oracles improvise concurrently, they act as independent agents. Some mechanisms should be provided for these agents to exchange information and take decisions. For instance if parallel oracles model different musical viewpoints, the prediction of the next value by e.g. the pitch oracle, could result in a constraint put on the duration oracle (not any duration may be accepted for a given pitch). In the case where multiple oracles model a polyphony of performers, they must have some way of synchronizing rythmically in order to deliver a consistent improvisation. This might pose a big challenge to the NTCC model since what is computed at each time unit might not be uniform for each viewpoint. Acting according

to information computed at, say, three time units in the past, might be awkward to model since the calculus only has "future" constructs.

- Feed back control :
  In order to implement reinforcement strategies, the system must be able to learn not only from the external world but also from the output of an oracle. A candidate sequence generated by an oracle could be modelled by a supervisor oracle (or antother model) in order to make some (e.g. entropy) measurements on it, and give a weight to the candidate. Certain transitions could be reinforced corrsponding to "good paths". This would be a way to control variety in the generation and to avoid falling into generaton loops. More powerful constructs than those that have been proposed in stochastic extensions of NTCC would be needed to model these kind of preferences for choosing some NTCC process for execution.

References

1. Assayag, G., Dubnov, S. "Using Factor Oracles for Machine Improvisation." *G. Assayag, V. Cafagna, M. Chemillier (eds.), Formal Systems and Music special issue*, Soft Computing 8, pp. 1432-7643, September 2004.
2. Dubnov, S., Assayag, G. "Universal Prediction Applied to Stylistic Music Generation" *in Mathematics and Music, A Diderot Mathematical Forum, Assayag, G.; Feichtinger, H.G.; Rodrigues, J.F. (Eds.)*, pp.147-160, Springer-Verlag, Berlin, 2002.
3. Dubnov, S., Assayag, G., El-Yaniv, R. "Universal Classification Applied to Musical Sequences" *Proc. Intl Computer Music Conf.*, Intl Computer Music Assoc., 1998, pp. 332-340.
4. Assayag, G., Dubnov, S., Delerue, O., "Guessing the Composers Mind: Applying Universal Prediction to Musical Style" *Proc. Intl Computer Music Conf.*, Intl Computer Music Assoc., pp. 496-499, 1999.
5. Conklin, D. "Music Generation from Statistical Models", *Proceedings of the AISB 2003 Symposium on Artificial Intelligence and Creativity in the Arts and Sciences*, Aberystwyth, Wales, 30 35, 2003.
6. Poirson, E., *Simulations d'improvisations l'aide d'un automate de facteurs et validation experimentale*, Rapport de DEA ATIAM, Universit Pierre et Marie Curie, 2002.
7. Dubnov, S., Assayag, G., Lartillot, O., Bejerano, G., "Using Machine-Learning Methods for Musical Style Modeling", *IEEE Computer*, Vol. 10, n 38, p.73-80, October 2003.
8. D. Ron, Y. Singer, and N. Tishby, "The Power of Amnesia: Learning Probabilistic Automata with Variable Memory Length", *Machine Learning*, vol. 25, 1996, pp. 117-149.
9. Allauzen C., Crochemore M., Raffinot M., " Factor oracle: a new structure for pattern matching" *Proceedings of SOFSEM'99, Theory and Practice of Informatics* J. Pavelka, G. Tel and M. Bartosek ed., Milovy, Lecture Notes in Computer Science 1725, pp 291-306, Springer-Verlag, Berlin, 1999.
10. C. Palamidessi and F. Valencia. "A Temporal Concurrent Constraint Programming Calculus" *Proc. of the Seventh International Conference on Principles and Practice of Constraint Programming CP2001*, Springer-Verlag, 2001.
11. C. Olarte and C. Rueda. "A Stochastic non-deterministic Temporal Concurrent Constraint Calculus" *Proc. of the XXXI Latinamerican Informatics Conference CP2005*, Cali, 2005.
12. C. Rueda and F. Valencia. "Proving musical properties Using a temporal Concurrent Constraints calculus" *Procedings of the ICMC2002*, Goteborg, Sweden, 2002.
13. V. Saraswat. *Concurrent Constraint Programming* The MIT Press, Cambridge, MA, 1993.
14. Assayag, G., Rueda C. , Laurson, M. Agon, C. Delerue, O. "Computer Assisted Composition at Ircam" *Computer Music Journal* 23:3, fall 1999