Formalizing Timed Musical Processes with a Temporal Concurrent Constraint Programming Calculus

Camilo Rueda and Frank D. Valencia

Universidad Javeriana-Cali, Colombia Email:`crueda@atlas.puj.edu.co`
BRICS, *, University of Aarhus, Denmark. Email:`fvalenci@brics.dk`

Abstract We propose the *ntcc calculus*, a model of *temporal concurrent constraint programming* with the capability of modeling asynchronous and non-deterministic timed behavior, as a convenient formalism for expressing temporal musical processes. We argue that ntcc provides a simple yet powerful framework for expressing a rich variety of musical events coordination. We illustrate its potential by formalizing in ntcc a musical improvisation process. We describe a proof system for linear temporal properties of ntcc processes and show its use for proving properties of musical processes.

1 Introduction

In the last decade several formalisms have been proposed to account for musical structures and the operations used to construct and transform them ([4], [3], [11]. Building up from these formalisms languages and tools for supporting the compositional task are then proposed ([2]). Usually these tools extend general purpose programming languages with new interfaces and capabilities which are deemed suitable to manipulate the chosen musical structures. The obvious advantage of this approach is that musicians can, at least in principle, use the expressiveness of the underlying language to tailor musical structures to fit particular compositional needs. A disadvantage is the distance between implementation language and music formalism that may result in lack of adaptation to the envisioned musical processes.

Our approach is different. We envision computer supported music composition from a model of computation perspective. We thus look for a reduced set of simple objects and behavior that should be a minimal base to construct meaningful musical processes. The base concept we choose to bootstrap from is that of concurrent process. We regard music performance and composition as a complex task of defining and controlling interaction among concurrent activities. We thus borrow concepts and techniques from concurrent processes modeling to define suitable computational calculi and analyze their behavior in real musical settings. What we gain from this *low level* approach is twofold. One the one hand, we are able to ground the development of music composition tools on a very precise formal foundation and by this means proposing coherent higher level musical structures and operations. On the other hand, our model can give us clues for constructing formal proofs of interesting properties of a given musical process.

In [1], *PiCO*, a concurrent processes calculus integrating constraints and objects was proposed. Musical applications are programmed in a visual language having this calculus as its underlying model. Since there is no explicit notion of time in *PiCO* some musical processes, in particular those involving real time activity, are difficult to express. Moreover, reasoning about musical processes behavior in *PiCO* can be difficult since there is no formal logic associated with it.

In this paper we propose using a temporal non deterministic concurrent calculus (*ntcc*, see [12]) as a formal base to model timed musical processes. The ntcc calculus inherits ideas from the tcc model [13], a formalism for reactive concurrent constraint programming. In tcc time is conceptually divided into *discrete intervals (or time-units)*. In a particular time interval, a

* Basic Research in Computer Science, Centre of the Danish National Research Foundation.

deterministic ccp process receives a stimulus (i.e. a constraint) from the environment, it executes with this stimulus as the initial store, and when it reaches its resting point, it responds to the environment with the resulting store. Also the resting point determines a residual process, which is then executed in the next time interval.

The tcc model is inherently deterministic and synchronous. Indeed, patterns of temporal behavior such as "the system must output c *within* the next t time units" or "the message must be delivered but *there is no bound* in the delivery time" cannot be expressed within the model. It also rules out the possibility of choosing one among several alternatives as an output to the environment.

A very important benefit of allowing the specification of non-deterministic and asynchronous behavior arises when modeling the interaction among several components running in parallel, in which one component is part of the environment of the others. This is frequent in musical applications. These systems often need non-determinism and asynchrony to be modeled faithfully.

The ntcc calculus is obtained from tcc by adding *guarded-choice* for modeling non-determinism and an *unbounded but finite delay* operator for asynchrony. Computation in ntcc progresses as in tcc, except for the non-determinism and asynchrony induced by the new constructs. The calculus allows for the specification of temporal properties, and for modeling and expressing constraints upon the environment both of which are useful in proving properties of timed systems. We shall illustrate the expressiveness of ntcc by modeling musical constructs such as rhythm patterns, and a musical process modeling controlled improvisation.

The logical nature of ntcc comes to the surface when we consider its relation with linear temporal logic: All the operators of ntcc correspond to temporal logic constructs. We use the sound system for linear temporal properties of ntcc given in [12] to prove properties of the modeled musical example.

The paper reports the main ideas of an ongoing project aiming at modeling in ntcc a wide variety of real rhythm musical processes.

2 The Calculus

In this section we describe the syntax and an operational semantics of the ntcc calculus. First we recall the notion of constraint system.

Basically, a constraint system provides a signature from which syntactically denotable objects in the language called *constraints* can be constructed, and an entailment relation specifying interdependencies between such constraints. The underlying language $\mathcal{L}$ of the constraint system contains the symbols $\dot{\neg}$, $\dot{\wedge}$, $\dot{\Rightarrow}$, $\dot{\exists}$, `true` and `false` which denote logical negation, conjunction, implication, existential quantification, and the always true and always false predicates, respectively. *Constraints*, denoted by $c, d, \ldots$ are first-order formulae over $\mathcal{L}$. We say that c *entails* d in Δ, written $c \vdash_\Delta d$ (or just $c \vdash d$ when no confusion arises), if $c \dot{\Rightarrow} d$ is true in all models of Δ. For operational reasons we shall require $\vdash$ to be decidable.

Process Syntax. Processes P, Q, $\ldots \in$ *Proc* are built from constraints $c \in \mathcal{C}$ and variables $x \in \mathcal{V}$ in the underlying constraint system by the following syntax.

$$\begin{array}{rl} P, Q, \ldots ::= & \mathbf{tell}(c) \ \mid \sum\limits_{i \in I} \mathbf{when}\, c_i \,\mathbf{do}\, P_i \mid P \parallel Q \ \mid \mathbf{local}\ x\ \mathbf{in}\, P \\ \mid & \mathbf{next}\, P \mid \mathbf{unless}\ c\ \mathbf{next}\, P \mid !\, P \quad \mid \star P \ . \end{array}$$

The only move or action of process $\mathbf{tell}(c)$ is to add the constraint c to the current store, thus making c available to other processes in the current time interval. The guarded-choice $\sum_{i \in I}$ **when** c_i **do** P_i, where I is a finite set of indexes, represents a process that, in the current time interval, must non-deterministically choose one of the P_j $(j \in I)$ whose corresponding constraint c_j is entailed by the store. The chosen alternative, if any, precludes the others. If no choice is possible then the summation is precluded. We use $\sum_{i\in I} P_i$ as an abbreviation for the "blind-choice" process $\sum_{i\in I}$ **when** (`true`) **do** P_i. We use **skip** as an abbreviation of the empty summation and "+" for binary summations.

Process $P \parallel Q$ represents the parallel composition of P and Q. In one time unit (or interval) P and Q operate concurrently, "communicating" via the common store. We use $\prod_{i\in I} P_i$, where I is finite, to denote the parallel composition of all P_i. Process **local** x **in** P behaves like P, except that all the information on x produced by P can only be seen by P and the information on x produced by other processes cannot be seen by P.

The process **next** P represents the activation of P in the next time interval. Hence, a move of **next** P is a unit-delay of P. The process **unless** c **next** P is similar, but P will be activated only if c cannot be inferred from the current store. The "unless" processes add (weak) time-outs to the calculus, i.e., they wait one time unit for a piece of information c to be present and if it is not, they trigger activity in the next time interval. We use $\mathbf{next}^n(P)$ as an abbreviation for $\mathbf{next}(\mathbf{next}(\ldots(\mathbf{next}\, P)\ldots))$, where **next** is repeated n times.

The operator ! is a delayed version of the replication operator for the $\pi-$calculus ([9]): $!\,P$ represents $P \parallel \mathbf{next}\, P \parallel \mathbf{next}^2 P \parallel \ldots$, i.e. unboundedly many copies of P but one at a time. The replication operator is the only way of defining infinite behavior through the time intervals.

The operator $\star$ allows us to express asynchronous behavior through the time intervals. The process $\star\, P$ represents an arbitrary long but finite delay for the activation of P. For example, $\star\,\mathbf{tell}(c)$ can be viewed as a message c that is eventually delivered but there is no upper bound on the delivery time. By using the $\star$ operator we can define a *fair asynchronous* parallel composition $P \mid Q$ as $(P \parallel \star\, Q) \; + \; (\star\, P \parallel Q)$. A move of $P \mid Q$ is either one of P or one of Q (or both). Moreover, both P and Q are eventually executed (i.e. a fair execution of $P \mid Q$).

Some Derived Constructs. We shall use $!_I P$ and $\star_I P$, where I is an interval of the natural numbers, as an abbreviation for $\prod_{i\in I} \mathbf{next}^i P$ and $\sum_{i\in I} \mathbf{next}^i P$, respectively. For instance, $\star_{[m,n]}P$ means that P is eventually active between the next m and $m+n$ time units, while $!_{[m,n]}P$ means that P is always active between the next m and $m+n$ time units.

Operational Semantics. Operationally, the current information is represented as a constraint $c \in \mathcal{C}$, so-called *store*. The operational semantics is given by considering transitions between *configurations* γ of the form $\langle P, c\rangle$. We define Γ as the set of all configurations.

We now informally describe the operational semantics of ntcc which is defined by reduction relations, as is usual in process calculi. The formal definition can be found in [12]. In ntcc two reduction relations are defined, one representing *internal transitions* and the other *observable transitions*. The *internal transition* $\langle P, c\rangle \longrightarrow \langle Q, d\rangle$ should be read as "P with store c reduces, in one internal step, to Q with store d ". The *observable transition* $P \xRightarrow{(c,d)} Q$ should be read as "P on input c reduces, in one time unit, to Q with store d ". The observable transition is defined in terms of a terminating sequence of internal transitions. The store does not transfer automatically from one interval to another.

To illustrate reductions in ntcc, consider a musical process that continually outputs either C (MIDI=60) or E (MIDI=64) until another process (the conductor) signals the end. Process $!\,P \;\parallel\; Q$, for P and Q as defined below models the example.

$$
\begin{aligned}
P &\stackrel{\text{def}}{=} (\mathbf{when}\,(go = 1)\,\mathbf{do}\,(\mathbf{tell}\,(note = 60) \;+\; \mathbf{tell}\,(note = 64))\,) \\
&\quad\; \parallel\; \mathbf{unless}\, end = 1\, \mathbf{next}\, \mathbf{tell}\,(go = 1) \\
Q &\stackrel{\text{def}}{=} \mathbf{tell}\,(go = 1) \;\parallel\; \star\,\mathbf{tell}\,(end = 1)
\end{aligned}
$$

Then

$\langle !\ P, go = 1\rangle \longrightarrow \langle P \ \| \ \mathbf{next}\ !\ P, go = 1\rangle \longrightarrow \langle \mathbf{tell}\ (note = 64) \ \| \ \mathbf{next}\ !\ P, go = 1\rangle$
$\longrightarrow \langle \mathbf{next}\ !\ P, note = 64 \wedge go = 1\rangle \not\longrightarrow$

Initially the store contains constraint $go = 1$ output by process Q. The replicated process $!P$ then creates a copy of P and schedules itself for the next time unit. Process P chooses note E (the store gets $go = 1 \wedge Notes = 64$). No further reductions are possible in the current time unit. Two processes, $!P$ and **tell** $go = 1$ are scheduled for the next time unit. One would then write $\langle !P \,\|\, Q, true\rangle \longrightarrow^* \langle \mathbf{next}\ !P \ \| \ \mathbf{next\ tell}\ go = 1, go = 1 \wedge note = 64\rangle \not\longrightarrow$ and also

$$!P \,\|\, Q \ \stackrel{(\mathbf{true}, go=1\wedge note=64)}{\Longrightarrow} \ !P \ \| \ \mathbf{tell}\ go = 1.$$

The first one is the internal transition relation, whereas the second is the observable transition. Reductions start all over again with these two processes in the next and subsequent time units, until process Q eventually puts in the store constraint $end = 1$.

2.1 Interpreting Runs of Processes.

Let us consider an infinite sequence of observable transitions

$$P \ = \ P_1 \ \stackrel{(c_1, c'_1)}{\Longrightarrow} \ P_2 \ \stackrel{(c_2, c'_2)}{\Longrightarrow} \ P_3 \ \stackrel{(c_3, c'_3)}{\Longrightarrow} \ \ldots$$

This sequence can be interpreted as a *interaction* between the system P and an environment. At the time unit i, the environment provides a *stimulus* c_i and P_i produces c'_i as *response*. If $\alpha = c_1.c_2.c_3.\ldots$ and $\alpha' = c'_1.c'_2.c'_3\ldots$, we represent the above interaction as $P \stackrel{(\alpha,\alpha')}{\Longrightarrow}{}^{\omega}$.

Alternatively, if $\alpha = \mathtt{true}^{\omega}$, we can interpret the run as an interaction among the parallel components in P without the influence of an external environment (i.e., each component is part of the environment of the others). In this case α is called the *empty* input sequence and α' is regarded as a *timed* observation of such an interaction in P.

As mentioned before, an important feature of the ntcc model is that there is a logic associated with it. We describe next this logic.

3 A Logic of ntcc Processes

A relatively complete formal system for proving whether or not an ntcc process satisfies a linear-temporal property was introduced in [12]. In this section we summarize these results.

The strongest postcondition of a process P, $sp(P)$, denotes the set of all infinite sequences that P can possibly output. More precisely,$sp(P) = \{\alpha' \mid \text{ for some } \alpha : P \stackrel{(\alpha,\alpha')}{\Longrightarrow}{}^{\omega}\}$.

Temporal Logic. We define a linear-temporal logic for expressing properties of ntcc processes. The formulae $A, B, ... \in \mathcal{A}$ are defined by the grammar $A ::= c \mid A \Rightarrow A \mid \neg A \mid \exists_x A \mid \circ A \mid \Box A \mid \Diamond A$. The symbol c denotes an arbitrary constraint. The symbols $\Rightarrow$, $\neg$ and $\exists_x$ represent linear-temporal logic implication, negation and existential quantification. These symbols are not to be confused with the logic symbols $\dot{\Rightarrow}$, $\dot{\neg}$ and $\dot{\exists}_x$ of the constraint system. The symbols $\circ$, $\Box$, and $\Diamond$ denote the linear-temporal operators *next*, *always* and *sometime*. We use $A \vee B$ as an abbreviation of $\neg A \Rightarrow B$ and $A \wedge B$ as an abbreviation of $\neg(\neg A \vee \neg B)$.

The standard interpretation structures of linear temporal logic are infinite sequences of states [8]. In ntcc states are represented with constraints, thus we consider as interpretations the elements of $\mathcal{C}^{\omega}$. When $\alpha \in \mathcal{C}^{\omega}$ is a model of A (see [12]) , we write $\alpha \models A$. We define $[\![A]\!] = \{\alpha \mid \alpha \models A\}$, i.e., the collection of all models of A.

We shall say that P *satisfies* A iff every infinite sequence that P can possibly output satisfies the property expressed by A, i.e., $sp(P) \subseteq [\![A]\!]$. A relatively complete proof system for assertions $P \ \vdash \ A$, whose intended meaning is that P satisfies A, can be found in [12]. We shall write $P \ \vdash \ A$ if there is a derivation of $P \ \vdash \ A$ in this system.

4 Musical applications

In this section we illustrate some ntcc examples. We first need to define an underlying constraint system.

Definition 1 (A Finite-Domain Constraint System). *Let max be a positive integer number. Define $FD[max]$ as the constraint system whose signature Σ includes symbols in $\{0, \mathsf{succ}, +, \times, =\}$ and the first-order theory Δ is the set of sentences valid in arithmetic modulo max.*

The intended meaning of $FD[max]$ is the natural numbers interpreted as in arithmetic modulo max. Henceforth, we assume that the signature is extended with two new unary predicate symbols call and change. We will designate Dom as the set $\{0, 1, ..., max - 1\}$ and use v and w to range over its elements.

Recursion. In ntcc it is possible to encode recursive definitions of the form $q(x) \stackrel{\text{def}}{=} P_q$, where q is the process name and P_q calls q at most once and such a call must be within the scope of a "**next**". Such a restriction keeps bounded the response time of the system. The intended behavior of a call $q(t)$, where t is a term fixed to a value v (i.e. $t = v$ in the current store), is that of $P_q[v/x]$, where $[v/x]$ is the operation of (syntactical) replacement of every occurrence of x by v. A precise encoding of recursion in ntcc is given in [12]. In what follows we rely on the usual intuitions concerning procedure calls in a programming language.

Cells. Cells provide a basis for the specification and analysis of mutable and persistent data structures. A *cell* can be thought of as a structure that contains a value, and if tested, it yields this value. A *mutable cell* is a cell that can be assigned a new value[1]. We model mutable cells of the form $x\colon(v)$, which we interpret as a variable x currently fixed to some v.

$$\begin{array}{lcl} x\colon(z) & \stackrel{\text{def}}{=} & \mathbf{tell}(x = z) \parallel \mathbf{unless}\ \mathsf{change}(x)\,\mathbf{next}\ x\colon(z) \\ exch_f(x, y) & \stackrel{\text{def}}{=} & \sum_v \mathbf{when}\ (x = v)\ \mathbf{do}\ (\,\mathbf{tell}(\mathsf{change}(x)) \parallel \mathbf{tell}(\mathsf{change}(y)) \\ & & \qquad \parallel\ \mathbf{next}(\,x\colon(f(v)) \ \parallel\ y\colon(v)\,)\ ). \end{array}$$

Definition $x\colon(z)$ represents a cell x whose current content is z. The current content of x will be the same in the next time interval unless it is to be changed next (i.e $\mathsf{change}(x)$). Definition $exch_f(x, y)$ represents an exchange operation between the contents of x and y. If v is x's current value then $f(v)$ and v will be the next x and $y's$ values, respectively. In the case of functions that always return the same value (i.e. constants), we will take the liberty of using that value as its symbol. For example, $x\colon(3) \parallel y\colon(5) \parallel exch_7(x, y)$ gives us the cells $x\colon(7)$ and $y\colon(3)$ in the next time interval. The assignment of v to a cell x, written $x := v$, can then be encoded as **local** y **in** $exch_v(x, y)$ where the local variable y is used as dummy variable (cell).

Rhythm patterns We can easily model synchronization of rhythm patterns in ntcc. Let us first define a "metronome" process.

$$\begin{array}{rl} M(i, count) \stackrel{\text{def}}{=} & \mathbf{tell}\ (tick = i) \\ & \parallel\ (\mathbf{when}\ count\%k = 0\ \mathbf{do}\ \mathbf{next}\ M(i+1, count+1) \\ & \parallel\ \mathbf{when}\ count\%k \neq 0\ \mathbf{do}\ \mathbf{next}\ M(i, count+1)\ ) \end{array}$$

Process $M(i, count)$ ticks every k time units. This process could be controlled by an acceleration process:

[1] A richer notion of cell can be found in ccp based models such as the Oz calculus [14], the π^+ calculus [6], and PiCO [1].

$$Accel(delay) \stackrel{\text{def}}{=} \textbf{tell}\ (k = delay)$$
$$\quad \parallel\ (\textbf{when}\ signal = 1\ \textbf{do next}\ Accel(delay \div 2)$$
$$\quad \parallel\ \textbf{when}\ signal = 0\ \textbf{do next}\ Accel(delay)$$

Process $Accel(delay)$ changes the ticking speed of M at each $signal$ of a controller. Now we can imagine rhythm processes synchronized by M and thus accelerated by a $Controller$ process:

$$Rhythm(start, dur, end) \stackrel{\text{def}}{=} \textbf{when}\ tick < start\ \textbf{do next}\ Rhythm(start, dur, end)$$
$$\quad \parallel\ \textbf{when}\ tick = start$$
$$\quad \textbf{do}\ (Pitch\ \parallel\ \textbf{unless}\ start + dur > end\ \textbf{next}\ Rythm(start + dur, dur, end)$$

Process $Rhythm(start, dur, end)$ runs a certain $Pitch$ process, which outputs some pitch of a note, each dur time units after $start$ during end time units. Adding rhythms of two eight notes, a triplet and a quintuplet can then be defined by the system:

$$System \stackrel{\text{def}}{=} \textbf{local}\ tick, signal\ \textbf{in}\ (Controller(signal)\ \parallel\ M(0,0)\ \parallel\ Accel(30)$$
$$\quad \parallel\ Rhythm(0,30,120)\ \parallel\ Rhythm(0,20,120)\ Rhythm(0,12,120)\ )$$

Since $rhythm$ process could independently change its relative speed dur, complex patterns of interactions of global and local speeds, such as metric modulations, can be modeled.

Controlled improvisation. In the following example we shall use the derived construct $E = \textbf{whenever}\, c\, \textbf{do}\, P$ which can be recursively defined as $\textbf{when}\, c\, \textbf{do} P\ \parallel\ \textbf{unless}\, c\, \textbf{next}\, E$. The construct E keeps waiting for c to hold, and if c holds at some time unit it executes P in that time unit.

This example models a controlled improvisation process. There is a certain number m of musicians (or $Voices$), each playing blocks of three notes. Each of them is given a particular pattern of allowed delays between each note in the block. The pattern can be obeyed in any order. For example, the pattern $[3,2,2]$ could be chosen to be interpreted as spacements of 2 then 3, then 2 between the notes. Once a block of three notes played, each musician must wait for the others to finish their respective blocks before start playing a new block. The exact time a new block will be started is not specified, but it is constrained to be no later than the sum of the durations of all patterns. For example, for three musicians and patterns $[3,2,2], [4,3,5]$ and $[3,3,4]$ no delay between blocks greater than 29 time units is allowed. Musicians keep playing notes in this manner until all of them play a note simultaneously. At this time the process ends.

In the model we use a note value of $\perp$ for a silence. Process $Voice_i$ models the activities of each musician. Process $Voice_i$ chooses a permutation of the pattern (tuple $[j,k,l]$) and, given this permutation, spawns processes to play a note at time j, but not before, then at time $j+k$ but not before, and finally at time $j+k+l$ (process $Choice$). After playing its block, each musician signals termination by setting cell $flag_i := 1$. The conductor then waits for all flags to be set, resets the flags and gives the signal $go = 1$ to start a new block, unless all voices have output a note at the same time. Notice that on receiving the go signal, $Voice_i$ eventually starts a new block no later than $delay$ which is the variable representing the total duration of the patterns. Constraint $c_i[note_i]$ denote asserting a note obeying some unspecified property.

$$Voice_i \stackrel{\text{def}}{=} \sum_{[j,k,l]\in I_i}(Choice^i_{[j,k,l]} \parallel \mathbf{next}^{j+k+l}(\ flag_i := 1 \parallel \mathbf{whenever}\,(go=1)\,\mathbf{do} \star_{[0,delay]}\ Voice_i))$$

$$\begin{aligned} Choice^i_{[j,k,l]} \stackrel{\text{def}}{=}\ & !_{[0,j-1]}\mathbf{tell}(note_i = \bot) \parallel \mathbf{next}^j\,\mathbf{tell}(c_i[note_i]) \\ & \parallel\ !_{[j+1,j+k-1]}\mathbf{tell}(note_i = \bot) \parallel \mathbf{next}^{j+k}\mathbf{tell}(c_i[note_i]) \\ & \parallel\ !_{[j+k+1,j+k+l-1]}\mathbf{tell}(note_i = \bot) \parallel \mathbf{next}^{j+k+l}\mathbf{tell}(c_i[note_i]) \end{aligned}$$

$$\begin{aligned} Conductor \stackrel{\text{def}}{=}\ & \mathbf{local}\ stop\ \mathbf{in}\ (\mathbf{tell}(stop=0) \\ & \parallel !\mathbf{when}\,(\textstyle\bigwedge_{i\in[1,m]} flag_i = 1) \wedge (stop = 0)\ \mathbf{do}\ \mathbf{tell}(go=1) \parallel \textstyle\prod_{i\in[1,m]} flag_i := 0 \\ & \parallel \mathbf{next}\ !\mathbf{when}\ \textstyle\bigvee_{i\in[1,m]} note_i = \bot\ \mathbf{do}\ \mathbf{tell}(stop=0)) \end{aligned}$$

$$Init \stackrel{\text{def}}{=} \textstyle\prod_{i\in[1,m]} c_i[note_i] \parallel \prod_{i\in[1,m]} flag_i : 0$$

$$System \stackrel{\text{def}}{=} Init \parallel \textstyle\prod_{i\in[1,m]} Voice_i \parallel Conductor$$

The system described above stops playing notes at all whenever the m voices play notes at the same time unit (i.e., $\bigwedge_{i\in[1,m]} note_i \neq \bot$). We can then use the logic for ntcc to formally specify and prove termination properties for this system. For example, we may wonder whether the assertion $System \vdash \Diamond \bigwedge_{i\in[1,m]} note_i \neq \bot$ holds. This assertion expresses that the system eventually stops playing notes at all regardless the voices' choices. We may also wonder whether there exists certain voices' choices for which the system stops playing note at all. For proving this we can verify whether the assertion $System \vdash \Box \bigvee_{i\in[1,m]} note_i = \bot$ does not hold, i.e., there is a run of the system for which at some time unit all the notes are different from $\bot$. From [12] we know that proving these termination properties can be formally proved in the deduction system for the ntcc logic described in [12]. Moreover, recent results on ntcc show that these kind of properties are indeed decidable, i.e., can be proven automatically.

5 Related and Future Work

We have described an ongoing project concerning modeling temporal musical processes with ntcc. The main advantages of this approach is that construction and manipulation of musical structures rests on the firm ground of a precise, simple, yet powerful "time aware" computational model. This allows us to better understand interactions among concurrent musical processes and thus having better clues for the development of coherent music composition tools, particularly for real time settings. A strong point of the described approach is that the linear temporal logic associated with ntcc can be used to prove (or disprove) interesting musical properties of processes before running them.

The tccp calculus [5] is the only other proposal for a non-deterministic timed extension of ccp that we know of. One major difference with ntcc is that the information about the store is carried through the time units, so the semantic setting is rather different. The notion of time is also different; in tccp each time unit is identified with the time needed to ask and tell information to the store. As for the constructs, unlike ntcc, tccp provides for arbitrary recursion and does not have an operator for specifying (unbounded) finite-delay. We think that the ntcc notion of time unit as the finest "grain" of time for a given system fits more naturally to musical applications.

We plan to pursue this line of work in three directions: first, modeling in ntcc a variety of metric modulations such as described in [10] and proving their properties. Second, extending ntcc to a probabilistic model following ideas in [7]. This is justified by the existence of rhythm patterns examples involving stochastic behavior which cannot be faithfully modeled with non-deterministic behavior. Third, we plan to define an abstract machine for ntcc and construct a music composition language on top of it.

References

1. G. Alvarez, J.F. Diaz, L.O. Quesada, C. Rueda, G. Tamura, F. Valencia, and G. Assayag. Integrating constraints and concurrent objects in musical applications: A calculus and its visual language. *Constraints*, January 2001.
2. Gérard Assayag, Camilo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer-assisted composition at ircam: From patchwork to openmusic. *Computer music journal*, 23(3):59–72, 1999.
3. M. Balaban and C. Samoun. Hierarchy, time and inheritance in music modelling. *Languages of Design*, 1(3):147–172, 1993.
4. Kablan Barbar, Anthony Beurivé, and Myriam Desainte-Catherine. Structures hierarchiques pour la composition assisté par ordinateur. In M. Chemillier and F. Pachet, editors, *Recherches et applications en informatique musicale*, Collection Informatique Musicale, pages 109–124, Hermes, Paris, 1998. HERMES.
5. F. de Boer, M. Gabbrielli, and M. C. Meo. A timed concurrent constraint language. *Information and Computation*, 1999. To appear.
6. J.F. Diaz, C. Rueda, and F. Valencia. A calculus for concurrent processes with constraints. *CLEI Electronic Journal*, 1(2), December 1998.
7. O. Herescu and C. Palamidessi. Probabilistic asynchronous pi-calculus. *FoSSaCS*, pages 146–160, 2000.
8. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems, Specification.* Springer, 1991.
9. R. Milner. *Communicating and Mobile Systems: the π-calculus.* Cambridge University Press, 1999.
10. Francois Nicolas. Le feuilleté du temps, essaie sur les modulations métriques. *Entretemps*, (9), 1990.
11. F. Pachet, G. Ramalho, and J. Carrive. Representing temporal musical objects and reasoning in the muses system. *Journal of new music research*, 25(3):252–275, 1996.
12. C. Palamidessi and F. Valencia. A temporal concurrent constraint programming calculus. In *Proc. of the Seventh International Conference on Principles and Practice of Constraint Programming*, 26 November 2001. Availabe via http://www.brics.dk/~fvalenci/publications.html.
13. V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, 4–7 July 1994.
14. G. Smolka. The Oz programming model. In Jan van Leeuwen, editor, *Computer Science Today*, Lecture Notes in Computer Science, vol. 1000, pages 324–343. Springer-Verlag, Berlin, 1995.