Concurrent Constraint Calculi: a Declarative Paradigm for Modeling Music Systems.

Carlos Olarte, Camilo Rueda and Frank D. Valencia

Introduction

Concurrent constraint programming (CCP) [25] has emerged as a simple but powerful paradigm for concurrent systems; i.e. systems of multiple agents that interact with each other as for example in a collection of music processes (musicians) performing a particular piece. A fundamental issue in CCP is the specification of concurrent systems by means of constraints. A constraint (e.g. $\text{note} > 60$) represents partial information about certain system variables. During the computation, the current state of the system is specified by a set of constraints ($\text{store}$). Processes can change the state of the system by telling (adding) information to the store and synchronize by asking information to the store.

The ntcc calculus [14] is a CCP formalism for modeling temporal reactive systems. The calculus is built upon a small number of basic ideas that capture several aspects of temporal nondeterministic behavior. In ntcc, processes can also be constrained by temporal requirements such as delays, time-outs and pre-emptions. Thus, the calculus integrates two dimensions of computation: a horizontal dimension dealing with partial information (e.g., $\text{note} > 60$) and a vertical one in which temporal requirements come into play (e.g., a process must be executed at any time within the next ten time units).

The above integration is remarkably useful for modeling complex musical processes, in particular for music improvisation. For example, for the vertical dimension one can specify that a given process can nondeterministically choose any note satisfying a given constraint. For the horizontal dimension one can specify that the process can nondeterministically choose the time to play the note subject to a given time upper bound. This nondeterministic view is particularly suitable for processes representing a musician’s choices when improvising. Similarly, the horizontal dimension may supply partial information on a rhythmic pattern that leaves room for variation while keeping a basic control, as in one of the examples given below.

We shall also illustrate how implementing a weaker ntcc model of a musical process may greatly simplify the formal verification of its properties. We argue that this modeling strategy provides a “runnable specification” for music problems that eases the task of formally reasoning about them.

Furthermore, ntcc is provided with a linear temporal process logic for verifying specifications of ntcc processes. We shall show how this logic gives a very expressive setting for formally proving the existence of interesting musical properties of a process. We shall also give examples of musical specifications in ntcc and use the linear temporal logic for proving properties of a realistic musical problem.
Preliminaries

CCP calculi are parametrized by a constraint system [25] which specifies the basic constraints agents can tell or ask during execution. A constraint represents a piece of information (or partial information) upon which processes may act. For instance, in a system with variables $\text{pitch}_1$, $\text{pitch}_2$ taking MIDI values, the constraint $\text{pitch}_1 > \text{pitch}_2$ specifies possible values for $\text{pitch}_1$ and $\text{pitch}_2$ (those where $\text{pitch}_1$ is at least a tone higher than $\text{pitch}_2$). The constraint system defines also an entailment relation ($\vdash$) specifying inter-dependencies between constraints. Intuitively, $c \vdash d$ means that the information $d$ can be deduced from the information represented by the constraint $c$. For example, $\text{pitch}_1 > 60 \vdash \text{pitch}_1 > 42$.

Formally, we can set up the notion of constraint system by using First-Order Logic as it was done in [27]. Let us suppose that $\Sigma$ is a signature (i.e., a set of constant, function and predicate symbols) and that $\Delta$ is a consistent first-order theory over $\Sigma$ (i.e., a set of sentences over $\Sigma$ having at least one model). Constraints can be thought of as first-order formulae over $\Sigma$. Consequently, the entailment relation is defined as follows: $c \vdash d$ if the implication $c \Rightarrow d$ is valid in $\Delta$. Notice that true is the smallest constraint in the sense that for any constraint $c$, $c \vdash\text{true}$. Analogously, false is the greatest constraint since from false any constraint can be deduced. This gives us a simple and general formalization of the notion of constraint system as a pair $(\Sigma, \Delta)$.

As an example, take the finite domain constraint system (FD) [11]. In FD, variables are assumed to range over finite domains and, in addition to equality, we may have predicates that restrict the possible values of a variable to some finite set. For instance, assume $x$ to be a FD variable with initial domain $\{1, 2, 3, \ldots, 10\}$. Adding the constraint $x > 5$ will restrict the domain of $x$ to be the set $\{6, 7, \ldots, 10\}$.

Alternatively, the notion of constraint system can be given in terms of Scott’s information system without consistency structure as it was done in [25].

CCP Processes

In the spirit of process calculi, the language of processes in the CCP model is given by a small number of primitive operators or combinators. A typical CCP process language features the following constructs:

- A tell operator adding a constraint to the store, thus making it available for other processes.
- An ask operator querying if a constraint can be deduced from the store.
- Parallel Composition combining processes concurrently.
- A hiding operator (also called restriction or locality) introducing local variables and thus restricting the interface a process can use to interact with others.

Following the notation in [14], we present the syntax of CCP in Definition 1.

**Definition 1 (CCP Processes)** Processes $P, Q, \ldots$ in CCP are built from constraints in the underlying constraint system by the following syntax:

$$P, Q := \text{skip} \mid \text{tell}(c) \mid \text{when } c \text{ do } P \mid P \parallel Q \mid (\text{local } \vec{x}; c) P$$
The process **skip** does nothing. It represents *inaction*. The process **tell(c)** adds the constraint *c* to the store. The process **when c do P** asks if *c* can be deduced from the store. If so, it behaves as *P*. In other case, it remains blocked until the store contains at least as much information as *c*. This way, ask processes define a synchronization mechanism based on entailment of constraints.

The process *P || Q* denotes the parallel composition of *P* and *Q*, i.e., *P* and *Q* running in parallel and possibly “communicating” via the common store.

Hiding on a set of variables *⃗x* is enforced by the process (**local ⃗x; c**)*P*. It behaves like *P*, except that all the information on the variables *⃗x* produced by *P* can only be seen by *P* and the information on the global variable in *⃗x* produced by other processes cannot be seen by *P*. The local information on *⃗x* produced by *P* corresponds to the constraint *c* representing a local store. We shall write (**local ⃗x**) as a shorthand for (**local ⃗x; true**) *P*.

**Timed CCP**

In CCP, the information in the store grows monotonically, i.e., once a constraint is added it cannot be removed. This condition has been relaxed by considering temporal extensions of CCP such as Timed CCP (**tcc**) [24]. In **tcc**, processes evolve along a series of *discrete time intervals* (or time units). Each interval contains its own store and information is not automatically transferred from one interval to another. In music, the duration of each time unit is related to the minimal metric value used in the piece.

**Definition 2 (Temporal Constructs)** The **tcc processes** result from adding in Definition 1 the following constructs

\[
P, Q \quad := \quad \text{next } P \mid \text{unless } c \text{ next } P \mid ! P
\]

The constructs above allow the processes in Definition 1 to have effect along the time units. The unit-delay **next P** executes *P* in the next time interval. The negative ask **unless c next P** is also a unit-delay but *P* is executed in the next time unit iff *c* is *not* entailed by the final store at the current time interval. This can be viewed as a (weak) time-out (or weak preemption): It waits one time unit for a piece of information *c* to be present and if it is not, it triggers activity in the next time interval. The process *P* must be guarded by a next process to avoid paradoxes such as a program that requires a constraint to be present at an instant only if it is not present at that instant (see [24]).

Notice that in general *Q = unless c next P* does not behave the same as *Q' = when ~c do next P*. This can be explained from the fact that *d ⊬ c* does not imply *d ⊨ ~c*. Take for example the final store *d = “x > 0”* and let *c = “x = 42”*. We have both, *x > 0 ⊬ x ≠ 42* and *x > 0 ⊨ x = 42*. Then, the process *P* is executed in *Q* but it is precluded from execution in *Q'*. Finally, the replication **!P** means *P || next P || next²P...*, i.e. unboundedly many copies of *P* but one at a time.

The **tcc model**. As we said before, processes in **tcc** evolve along a series of discrete time intervals. In each time interval, a deterministic CCP processes receives a stimulus (i.e. a constraint) from the environment (an input to the system). It executes with this stimulus as the *initial store*. When it reaches its resting point, i.e., no further evolution is
possible, it responds to the environment with the final store (the output of the system).
Furthermore, the resting point determines a residual process, which is then executed in
the next time interval. Roughly speaking, the residual process is obtained by “unfolding”
the sub-terms within “next” and “unless” expressions (see [14]).

This view of reactive computation is particularly appropriate for programming reactive
systems in the sense of Synchronous Languages [3], i.e., systems that react continuously
with the environment at a rate controlled by the environment.

Extensions to Timed CCP

Several extensions of the basic constructs presented above have been studied in the literature
in order to provide settings for the programming and specification of systems with the
declarative flavor of concurrent constraint programming. For example, temporal exten-
sions to deal with reactive systems have been introduced in [24, 6, 14], non-deterministic
behavior and asynchrony [14, 6], probabilistic behavior [9, 4, 16, 20], mobility [18], lin-
earity [7], etc. We shall briefly describe some of these extensions (see Figure 1).

Non-determinism and Asynchrony.

The tcc calculus is a model of reactive computation based on the idea of Synchronous
Languages [3], i.e., programs must be determinate and respond immediately to input
signals. Therefore, it is not meant for modeling nondeterministic or asynchronous tem-
poral behavior. Indeed, patterns of temporal behavior such as “the system must output c

Figure 1. Hierarchy of CCP-based languages.
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.

The $\text{ntcc}$ calculus [14] is obtained by adding guarded-choices for modeling non-deterministic behavior and an unbounded finite-delay operator for asynchronous behavior. Computation in $\text{ntcc}$ progresses as in $\text{tcc}$, except for the nondeterminism induced by the new constructs.

**Definition 3 ($\text{ntcc}$ Processes)** The $\text{ntcc}$ processes result from adding to the syntax in Definition 2 the following constructs:

$$\sum_{i \in I} \text{when } c_i \text{ do } P_i \mid \star P$$

The guarded-choice $\sum_{i \in I} \text{when } c_i \text{ do } P_i$, where $I$ is a finite set of indices, represents a process that, in the current time interval, must non-deterministically choose one of the $P_j$ ($j \in I$) whose corresponding guard (constraint) $c_j$ is entailed by the store. The chosen alternative, if any, precludes the others. If no choice is possible then the summation remains blocked until more information is added to the store.

The operator “$\star$” corresponds to the unbounded but finite delay operator $\epsilon$ for synchronous CCS [13] and it allows to express asynchronous behavior through the time intervals. Intuitively, process $\star P$ represents $P + \text{next } P + \text{next }^2 P + \ldots$, i.e., an arbitrary long but finite delay for the activation of $P$.

**Strong Pre-Emption in $\text{tcc}$**

The work in [28] introduces the notion of strong pre-emption in $\text{tcc}$, i.e. the time-out operations can trigger activity in the current time interval. Strong pre-emption is useful when an action must be triggered immediately on the absence of a constraint $c$ rather than delayed to the next interaction with the environment as in $\text{unless } c \text{ next } P$. In this case, it is assumed that $c$ has not been produced in the store, and will not be produced throughout system execution at the present time instant.

**Definition 4 (Default $\text{tcc}$ Processes)** The Default $\text{tcc}$ processes result from adding in the syntax in Definition 2 the following construct:

$$\text{if } c \text{ else } P$$

Intuitively, if $c$ cannot be deduced from the current store now and it will not be produced during the current time interval, the process $P$ is executed. To avoid paradoxes, it is assumed that the process $P$ does not add $c$ to the current store (see [28]).

**Stochastic and Probabilistic Behavior**

When modeling a complex system, it is no longer possible to assume that enough information will be provided for all its components. Hence, models have to deal with uncertain
behavior or an incomplete description of these components. A natural model is then to consider these components as stochastic ones where a probability distribution is given to the set of outputs partially known components may perform. Indeed, apart from providing an intuitive way of describing alternative behaviors, probability distributions often ease the integration of statistic and empirical data into models.

In [8], the authors develop Probabilistic CCP (pcc) and its temporal extension ptcc to model synchronous reactive probabilistic systems. In these languages, (discrete) random variables with a given probability distribution are introduced. A run of the system will choose a value for the random variables with the given probability; these probability values accumulate as more and more choices are made in the course of the run. Alternate choices lead to alternate runs, with their own accumulated probability values.

**Definition 5 (pcc and ptcc Processes)** The pcc (resp. ptcc) processes result from adding to the syntax in Definition 1 (resp. Definition 2) the process

\[ \text{new}(r,f) \text{ in } P \]

where \( r \) is a variable and \( f \) is its probability mass function.

Intuitively, the process \( \text{new}(r,f) \text{ in } P \) chooses a value for \( r \) according to the function \( f \) and then execute \( P \).

The authors in [20] study also the integration of probabilistic information into CCP. The language pntcc is then proposed featuring both non-deterministic and probabilistic behavior. The operational semantics of pntcc ensures the consistent interactions between both the probabilistic and the non-deterministic choices. The semantics is based on a probabilistic automaton [26] that separates the internal choices made probabilistically by the processes from those external choices made non-deterministically under the influence of a scheduler. As a result, the observable behavior of a system —what the environment perceives from its execution— formalized by the semantics is purely probabilistic; the influence of non-determinism is regarded as unobservable.

In the CCP spirit, pntcc allows to reason about process specifications in terms of properties including explicit quantitative information: pntcc processes are related to formulae in the probabilistic logic PCTL [10]. This relation is based on the fact that the observable behavior of a process can be interpreted as the discrete time Markov chain (DTMC) defining satisfaction in PCTL.

**Definition 6 (pntcc Processes)** The pntcc processes result from adding to the syntax in Definition 2 the process

\[ \bigotimes_{i \in I} \text{when } c_i \text{ do } (P_i, a_i) \]

where \( I \) is a finite set of indices, and for every \( a_i \in \mathbb{R}^{(0,1]} \) we have \( \sum_{i \in I} a_i = 1 \)

The intuition of this operator is as follows. Each \( a_i \) associated with the process \( P_i \) represents its probability of being selected for execution. Hence, the collection of all \( a_i \) represents a probability distribution.
Mobile Behavior

In [18] the authors introduce Universal Timed CCP (utcc), a calculus aiming at modeling mobile reactive systems, i.e., systems that can change their communication structure while interacting continuously with their environment. Here we understand mobility as in the π-calculus [21], i.e. generation and communication of private channels or links.

The basic move from tcc to utcc is to replace the ask operation \( \text{when } c \text{ do } P \) by a temporary parametric ask constructor of the form \((\text{abs } \vec{x}; c) P\). This process can be viewed as an abstraction of the process \(P\) on the variables \(\vec{x}\) under the constraint (or with the guard) \(c\). Intuitively, \((\text{abs } \vec{x}; c) P\) performs \(P[\vec{t}/\vec{x}]\) in the current time interval for all the terms \(\vec{t}\) s.t. \(c[\vec{t}/\vec{x}]\) is entailed by the store. The abstraction construct in utcc has a pleasant duality with the local operator. From a programming language perspective, the variables \(\vec{x}\) in \((\text{local } \vec{x}; c) P\) can be seen as the local variables of \(P\) while \(\vec{x}\) in \((\text{abs } \vec{x}; c) P\) as the formal parameters of \(P\).

**Definition 7 (utcc Processes)** The utcc processes result from replacing in the syntax in Definition 2 the expression \(\text{when } c \text{ do } P\) with

\[(\text{abs } \vec{x}; c) P\]

where variables in \(\vec{x}\) are pairwise distinct.

When the set of variables \(\vec{x}\) in \((\text{abs } \vec{x}; c) P\) is empty, we retrieve the classical ask operator \(\text{when } c \text{ do } P\).

The extra expressiveness of utcc to model mobile behavior has found application, e.g., to model dynamic music systems where the composer can dynamically change the hierarchical structure of the score according to the information derived from the environment [15].

Logic Characterization of CCP

CCP calculi enjoys a declarative nature that distinguishes it from other models of concurrency: CCP programs (or processes) can be seen, at the same time, as both computing agents and logic formulae (see e.g., [25, 14, 5, 17]), i.e., programs can be read and understood as logical specifications. A natural benefit of this alternative view is to provide the modeler or developer with a language suitable for both the specification and implementation of programs. Let us elaborate on this ideas in the context of the ntcc calculus. See [14] for a detailed treatment on this topic.

Linear temporal logics (LTL) have been extensively used to specify properties of timed systems [12]. Formulae in this logic are built from the following syntax:

**Definition 8 (Logic Syntax)** The formulae \(A, B, ... \in \mathcal{A}\) are defined by the grammar

\[A, B, ... ::= c \mid A \Rightarrow A \mid \neg A \mid \exists x A \mid \circ A \mid \square A \mid \diamond A\]

Here \(c\) denotes an arbitrary constraint which we shall refer to as atomic proposition. The intended meaning of the other symbols is the following: \(\Rightarrow\) and \(\exists\) represent linear-temporal logic implication, negation and existential quantification. These symbols are not to be confused with the symbols \(\Rightarrow\), \(\neg\) and \(\exists\) of the underlying constraint system.
The symbols ◦, □, and ◊ denote the temporal operators next, always and sometime. Intuitively ◦A, ◊A and □A means that the property A must hold next, eventually and always, respectively.

Let P be a process and A a LTL formulae specifying a given temporal property. One may wonder whether the process P satisfies the specification A, written as P |= A. The intended meaning of this assertion is that, regardless the input, every output of P satisfies the temporal formula A.

Let us give some examples. Since in every infinite sequence output by ◦tell(c) on arbitrary inputs there must be an element entailing c, we have ◦tell(c) |= ◊c. Analogously, !tell(c) |= □c since all output of P must entail c. Let P = tell(c) + tell(d). We have P |= (c ⊕ d) as every constraint e output by P entails either c or d. Finally, when c do next tell(d) |= c ⇒ ◦d since the entailment of c in the first time unit implies the entailment of d in the next one.

**Representation of time**

As described above, in CCP calculi time is usually thought of in terms of state transitions. Conceptually, a computation instantly produces some information and then reaches a quiescent state. The information output is what is observable in that state. Besides information, as a result of the computation some processes might be scheduled for a “future time”. This “future time” refers to a particular new state where the information the process computes is observable. In ntcc, a time unit is thus related to a state. Two different states refer to different times, but there is no explicit notion of distance between the two states. In practice one might think the extension of each time unit to represent the smallest temporal separation between observable changes in the system. The analogy in music would be the smallest rhythmical unit in a score, with no “absolute” temporal duration involved. Temporal models in ntcc would then typically “synchronize” so that information changes conforms to multiples of the smallest unit. The ntcc interpreter would then have to give some actual absolute duration to this unit, like a conductor giving tempo.

For the simple score shown below

a ntcc model could be

\[ P \overset{\text{def}}{=} \text{tell}(\text{note}_1 = 72) \parallel \text{next}^3(\text{tell}(\text{note}_2 = 75) \parallel \text{next} (\text{tell}(\text{note}_3 = 70) \parallel \text{next}^4(\text{tell}(\text{note}_4 = 70))) \]

where each time unit is seen to correspond to a sixteenth note. Now, it is up to the implementation (i.e. the interpreter) to ensure that each time unit takes exactly whatever actual absolute duration is desired for the sixteenth note. A more realistic model would take advantage of the synchronizing capabilities of the calculus to associate each note to
a process deciding when to play by looking at a conductor:

\[
P_1 = \text{when } \text{signal} = 0 \text{ do tell}(\text{note}_1 = 72) \parallel \text{unless } \text{signal} = 0 \text{ next } P_1 \\
P_2 = \text{when } \text{signal} = 3 \text{ do tell}(\text{note}_2 = 75) \parallel \text{unless } \text{signal} = 3 \text{ next } P_2 \\
P_3 = \text{when } \text{signal} = 4 \text{ do tell}(\text{note}_3 = 70) \parallel \text{unless } \text{signal} = 4 \text{ next } P_3 \\
P_4 = \text{when } \text{signal} = 8 \text{ do tell}(\text{note}_4 = 70) \parallel \text{unless } \text{signal} = 8 \text{ next } P_4
\]

The score is then defined by a process \( Q \)

\[
Q = P_1 \parallel P_2 \parallel P_3 \parallel P_4
\]

In this case the interpretation of the absolute duration of the sixteenth note is left to a signal from the environment (say supplied by a Max-MSP process [22]) or from a \texttt{ntcc} process modeling the conductor.

Let us now consider each of the above musical process with respect to some simple temporal operations. Suppose corresponding \texttt{ntcc} definitions for process \( R \) modeling

Now, we would like to model

In the first representation this is done by process

\[
R \parallel \text{next}^{16} P
\]

whereas in the second the conductor would have to use two hands, providing \textit{signal}_1 for \( R \) and \textit{signal} for \( Q \). In general, each process should use its own different signal and the conductor should be aware of that. Moreover, this is only a partial solution since it would not work for two or more instances of, say, process \( P \). When processes are combined as voices going simultaneously, e.g.,

parallel composition \( P \parallel R \) works appropriately in both representations. However, a slight metric transformation such as
cannot be expressed as some composition of $P$ and $R$, but it could, in principle, still be modeled as $Q \parallel R$ assuming the conductor accelerates the pace of signal while maintaining that of $\text{signal}_1$.

Timed CCP calculi thus provide a logical but not a metrical view of time. In particular, no support for metric combinations of the temporal occurrence of processes is given and it is up to the user to devise coordination patterns to represent metric structures in some way. Devising schemes to overcome this limitation while keeping the clean semantics of tcc is one of our aims in the future. One can envision combinators such as

<table>
<thead>
<tr>
<th>combinator</th>
<th>meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \downarrow_k Q$</td>
<td>$P$ followed by $Q$ delayed $k$ units</td>
</tr>
<tr>
<td>$P \ll k$</td>
<td>process $P$ stretched proportionally $k$ units</td>
</tr>
<tr>
<td>$P \gg k$</td>
<td>process $P$ expanded proportionally $k$ units</td>
</tr>
<tr>
<td>$P \triangledown Q$</td>
<td>process $Q$ time-intercalated with $P$</td>
</tr>
</tbody>
</table>

and others.

When real-time behavior of models is intended, there is the issue of taking explicitly into account the time needed for a computation. That is, using a more fine grained notion of time in which the “duration” of a state of affairs is determined by the time spent in doing the computation defined for that state. In this view, a state could no longer be defined by whatever information is output in a time unit as a whole (considered thus as instantaneous), but instead by each action over the store. Each time unit would be composed of several such states. In this view, a state corresponds to the information contained in the store. Each change in the store defines a new state. What would then be the time elapsed between two such states? Since the only possible change in a store is the addition of a constraint, a natural proposal would be to consider the “distance” between states $s_1$ and $s_2$ to be that constraint differentiating them. Whether this provides a coherent metric in practice is something we plan to dwell in the future.

**Musical Applications**

In the last decade several formalisms have been proposed to account for musical structures and the operations used to construct and transform them [2, 1, 19]. We can regard music performance and composition as a complex task of defining and controlling interaction among concurrent activities. In [23], PiCO, a concurrent process 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 examples, in particular those involving time and synchronization, are difficult to express. In this section we model two of those examples.

In the examples below we shall use the derived operators $\star_{[m,n]}P$, $\cdot_{[m,n]}P$ and $W_{(c,P)}$. The process $\star_{[m,n]}P$ means that $P$ is eventually active exactly once between the next $m$ and $m + n$ time units, while $\cdot_{[m,n]}P$ means that $P$ is always active between the next $m$ and $m + n$ time units. The construct $W_{(c,P)}$ waits until $c$ holds and then executes $P$:

$$ W_{(c,P)} \overset{\text{def}}{=} \text{when } c \text{ do } P \parallel \text{unless } c \text{ next } W_{(c,P)}. $$

We shall use the more readable notation $\text{wait } c \text{ do } P$. Finally, to specify mutable and persistent data structures we shall use cells. Let us assume that the signature of the underlying constraint system is extended with an unary predicate symbol change. A mutable cell $x(v)$ can be viewed as a structure $x$ which has a current value $v$ and which can, in the future, be assigned a new value.

$$ x: (z) \overset{\text{def}}{=} \text{tell}(x = z) \parallel \text{unless } \text{change}(x) \text{ next } x: (z) $$

Intuitively, definition $x: (z)$ represents a cell $x$ whose value is $z$ and it will be the same in the next time interval unless it is to be changed next (i.e., change($x$)). Let $\text{exch}_y[x,y]$ be defined as

$$ \text{exch}_y[x,y] \overset{\text{def}}{=} \sum_v \text{when } x = v \text{ do } ( \text{tell}(\text{change}(x)) \parallel \text{tell}(\text{change}(y)) \parallel \text{next} (x: (g(v)) \parallel g: (v) ) ) $$

This process represents an exchange operation in the following sense: if $v$ is $x$’s current value then $g(v)$ and $v$ will be the next values of $x$ and $y$ respectively.

**Controlled Improvisation.**

This example models a controlled improvisation musical system. Such a system can be described as follows: There is a certain number $m$ of musicians (or voices), each playing blocks of three notes. Each of them is given a particular pattern (i.e., a list) of allowed delays between each note in the block. The musician can freely choose any permutation of his pattern. For example, given a pattern $p = [4,3,5]$ a musician can play his block with spaces of 5 then 4 and then 3 between the notes. Once a musician has finished playing his block of three notes, he must wait for a signal of the conductor telling him that the others musicians have also finished their respective blocks. Only after this he can start playing a new block. The exact time in which he actually starts playing a new block 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 $p_1 = [3,2,2]$, $p_2 = [4,3,5]$ and $p_3 = [3,3,4]$ no delay between blocks greater than 29 time units is allowed. The musicians keep playing this way until all of them play a note at the same time. After this, all the musicians must stop playing.

In order to model this example we assume that constant $\text{sil} \in D$ represents some note value for silence. The process $M_i$, $i \leq m$, models the activity of the $i$–th musician. When ready to start playing ($\text{start}_i = 1$), the $i$–th musician chooses a permutation $(j,k,l)$ of his given pattern $p_i$. Then, $M_i$ spawns a process $\text{Play}_{(j,k,l)}$, thus playing a note at time $j$ (after starting), but not before, then at time $j + k$ but not before, and
finally at time \( j + k + l \). Constraint \( c_i[\text{note}_i] \) specifies some value for \( \text{note}_i \), different from \( \text{sil} \). After playing his block, the \( i \)-th musician signals termination by setting cell \( \text{flag}_i \) to 1. Furthermore, upon receiving the \( \text{go} = 1 \) signal, the \( i \)-th musician eventually starts a new block no later than \( \text{pdur} \) which is a constant representing the sum of the durations of all patterns.

\[
M_i \overset{\text{def}}{=} \text{when } (\text{start}_i = 1) \text{ do } \sum_{(j,k,l) \in \text{perm}(p_i)} (\text{Play}_{(j,k,l)}^i) \parallel \text{next}^{j+k+l}(\text{flag}_i := 1) \parallel \text{wait}(\text{go} = 1) \parallel \text{tell}(\text{start}_i = 1) \text{ end}
\]

\[
\text{Play}_{(j,k,l)}^i \overset{\text{def}}{=} \text{tell}(\text{note}_i = \text{sil}) \parallel \text{next}^j \text{tell}(c_i[\text{note}_i]) \parallel \text{tell}(\text{note}_i = \text{sil}) \parallel \text{next}^{j+k} \text{tell}(c_i[\text{note}_i]) \parallel \text{tell}(\text{note}_i = \text{sil}) \parallel \text{next}^{j+k+l} \text{tell}(c_i[\text{note}_i])
\]

Here \( \text{flag} \) is a mutable cell (see definition above) and \( \text{flag} := 1 \) represents the assignment of 1 to \( \text{flag} \).

The \( \text{Conductor} \) process is always checking (listening) whether all the musicians play a note exactly at the same time \( \bigwedge_{i \in [1,m]} (\text{note}_i \neq \text{sil}) \). If this happens it sets the cell \( \text{stop} \), initially set to 0, to 1. At the same time, it waits for all flags to be set to 1, and then resets the flags and gives the signal \( \text{go} = 1 \) to all musician to start a new block, unless all of them have output a note at the same time (i.e., \( \text{stop} = 1 \)).

\[
\text{Conductor} \overset{\text{def}}{=} \text{wait} \bigwedge_{i \in [1,m]} (\text{note}_i \neq \text{sil}) \text{ do } \text{stop} := 1 \text{ end}
\]

\[
\text{Conductor} = \text{wait} \bigwedge_{i \in [1,m]} (\text{note}_i \neq \text{sil}) \text{ do } \text{stop} := 1 \text{ end}
\]

\[
\text{System} = \text{Init} \parallel \text{Conductor} \parallel \prod_{i \in [1,m]} M_i
\]

Initially the \( m \) flag cells are set to 0, the \( M_i \) are given the start signal \( \text{start}_i = 1 \) and, as mentioned above, the cell \( \text{stop} \) is set to 0. The system (i.e., the performance) \( \text{System} \) is just the parallel execution (performance) of all the \( M_i \) musicians controlled by the \( \text{Conductor} \) process.

\[
\text{Init} \overset{\text{def}}{=} \prod_{i \in [1,m]} (\text{flag}_i : 0 \parallel \text{tell}(\text{start}_i = 1)) \parallel \text{stop} : 0
\]

The temporal logic and the proof system of \text{ntcc} [14] can then be used to formally specify and prove termination properties for this system. For example, we may wonder whether the assertion

\[
\text{System} \models \Diamond \text{stop} = 1
\]

holds. This assertion expresses that the musicians eventually stop playing at all regardless their choices. We may also wonder whether there exists certain choices of musicians for which they eventually stops playing note at all. For proving this we can verify whether the assertion

\[
\text{System} \models \Box \text{stop} = 0
\]
does not hold, i.e., there is a run of the system for which at some time unit all the notes are different from \texttt{sil}.

**Rhythm Patterns**

In this section we shall model synchronization of rhythm patterns in \texttt{ntcc}. Let us first define a “metronome” process.

\[
M[tick, count, \delta] \overset{\text{def}}{=} ! ( \text{when (count mod } \delta) = 0 \text{ do tick := tick + 1 || count := 0} \\
+ \text{when (count mod } \delta) > 0 \text{ do count := count + 1)}
\]

One could think of \(M[tick, count, \delta]\) as a process that “ticks” (by increasing \(\text{tick}\)) every \(\delta\) time units. This process could be controlled by the acceleration process:

\[
\text{Accel}[signal, \delta] \overset{\text{def}}{=} ! \text{ when signal = 1 } \wedge \delta > 0 \text{ do } \delta := \delta - 1
\]

The process \(\text{Accel}[signal, k]\) can “speed up the ticks of \(M[tick, count, \delta]\)” by decreasing \(\delta\), if some other process, which we shall refer to as \(\text{Control}[signal]\), tells \(\text{signal = 1}\).

We can now define the Rhythm process \(R(s,d,e)[tick, note]\) which can be synchronized by \(M[tick, count, \delta]\) and thus possibly accelerated by \(\text{Control}[signal]\).

\[
R(s,d,e)[tick, note] \overset{\text{def}}{=} ! \text{ when } s \geq \text{tick} \geq e \wedge (\text{tick - start}) \mod d = 0 \text{ do Pitch}[note]
\]

The process \(R(s,d,e)[tick, note]\) runs a certain \(\text{Pitch}[note]\) process, which outputs some pitch on \(\text{note}\), at every \((d)\)uration-th tick, from the \((s)\)tart-th tick to the \((e)\)nd-th tick. Adding rhythms of two eight notes, two triplets and two quintuplets can then be defined by the system:

\[
\text{System} \overset{\text{def}}{=} \text{(local tick } \delta \text{ count signal) ( Init } \parallel \text{ Control}[signal]} \parallel M[tick, count, \delta] \parallel \text{Accel}[signal, \delta] \parallel R(0,30,120)[tick, \delta] \parallel R(0,20,120)[tick, \delta] \parallel R(0,12,120)[tick, \delta]
\]

where \(\text{Init} = \text{tick} : 0 \parallel \delta : 30 \parallel \text{count} : 0\).

Since the \textit{Rhythm} processes depend on variables \textit{tick} and \(\delta\), complex patterns of interactions of global and local speeds, such as metric modulations, can be modeled.

**Future work**

A strong asset of the declarative paradigm we described in this chapter is the fact that a model of a system is itself a simulator. For this advantage to be put into practice efficient interpreters must be provided. Although the authors have devised interpreters for several of the \texttt{tcc}-like calculi described above, more work is needed in interpretation strategies to ensure efficiency, in particular for real-time settings.

As explained above, the declarative feature of CCP calculi allows to formally reason about system properties. It is thus possible to guarantee the existence of fundamental musical properties in a model, without recurring to a possibly very time-consuming
testing process. Automatic verification tools should be developed to ease the task of performing such proofs for complex models.

We have argued how the use of constraints greatly simplifies the task of devising synchronization patterns between processes. Since our aim is to take advantage of this fact to propose the CCP paradigm for complex music system modeling, we need to supply CCP calculi with more sophisticated temporal combinators, as described above. The challenge is to find ways in which these new constructs can find a place in the calculus and still maintain its clean semantics and declarative flavor.

Real-time behavior is at the core of many music systems, such as improvisation. Giving the user the right intuitions into the fine-grained temporal behavior of a model to achieve real-time awareness poses both a theoretical and implementation challenge. The theoretical aspects deals with finding the right calculus semantics for expressing concepts such as “true concurrence”, “simultaneity” and “temporal distance”. On the implementation side, the challenge is to couple this with an efficient interpreter with effective user temporal-controls.

Bibliography


