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

Carlos Olarte, Camilo Rueda and Frank D. Valencia

Introduction

Concurrent constraint programming (CCP) [24] has emerged as a simple but powerful paradigm for *concurrent systems*; i.e. systems of multiple agents that interact with one another 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. $note > 60$) represents partial information about certain system variables. During the computation, the current state of the system is specified by a set of constraints (*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 few basic ideas but it captures 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., $note > 60$) 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 that the use the `ntcc` calculus to build weaker representations of musical processes that greatly simplifies the formal expression and analysis 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 parametric in a *constraint system* [24] which specifies the basic constraints agents can tell or ask during execution. A constraint represents then a piece of information (or partial information) upon which processes may act. For instance, in a system with variables $pitch_1$, $pitch_2$ taking MIDI values, the constraint $pitch_1 > pitch_2$ specifies possible values for $pitch_1$ and $pitch_2$ (those where $pitch_1$ is at least a tone higher than $pitch_2$). 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, $pitch_1 > 60 \vdash pitch_1 > 42$.

Formally, we can set up the notion of constraint system by using First-Order Logic as it was done in [26]. Let us suppose that Σ is a signature (i.e., a set of constant, functions and predicate symbols) and that Δ is a consistent first-order theory over Σ (i.e., a set of sentences over Σ having at least one model). Constraints can be thought of as first-order formulae over Σ. Consequently, the entailment relation is defined as follows: $c \vdash d$ if the implication $c \Rightarrow d$ is valid in Δ. Notice that `true` is the smallest constraint in the sense that for any constraint c, $c \vdash$ `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 (Σ, Δ).

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 $\{1, 2, 3, ..., 10\}$. Adding the constraint $x > 5$ will restrict the domain of x to be the set $\{6, 7, ..., 10\}$.

Alternatively, the notion of constraint system can be given in terms of Scott's information system without consistency structure as it was done in [24].

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:*

$$P, Q := \mathbf{skip} \mid \mathbf{tell}(c) \mid \mathbf{when}\ c\ \mathbf{do}\ P \mid P \parallel Q \mid (\mathbf{local}\,\vec{x}; c)\,P$$

The process **skip** does nothing. It represents *inaction*. The process $\mathbf{tell}(c)$ adds the constraint c to the store. The process $\mathbf{when}\ c\ \mathbf{do}\ P$ *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 $P \parallel Q$ 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 $\vec{x}$ is enforced by the process $(\mathbf{local}\,\vec{x}; c)\,P$. It behaves like P, except that all the information on the variables $\vec{x}$ produced by P can only be seen by P and the information on the global variable in $\vec{x}$ produced by other processes cannot be seen by P. The local information on $\vec{x}$ produced by P corresponds to the constraint c representing a *local store*. We shall write $(\mathbf{local}\,\vec{x})\,P$ as a shorthand for $(\mathbf{local}\,\vec{x}; \mathtt{true})\,P$.

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 `tcc` [23]. In `tcc`, processes evolve along a series of *discrete time intervals*. Each interval contains its own store and information is not automatically transferred from one interval to another.

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

$$P, Q \quad := \quad \mathbf{next}\,P \mid \mathbf{unless}\ c\ \mathbf{next}\,P \mid !\,P$$

The constructs above allow the processes in Definition 1 to have effect along the time units. The unit-delay $\mathbf{next}\,P$ executes P in the next time interval. The *negative ask* $\mathbf{unless}\ c\ \mathbf{next}\,P$ 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: 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 [23]).

Notice that in general $Q = \mathbf{unless}\ c\ \mathbf{next}\,P$ does not behave the same as $Q' = \mathbf{when}\ \neg c\ \mathbf{do}\ \mathbf{next}\,P$. This can be explained from the fact that $d \nvdash c$ does not imply $d \vdash \neg c$. Take for example the final store $d =$ "$x > 0$" and let $c =$ "$x = 42$". We have both, $x > 0 \nvdash x \neq 42$ and $x > 0 \nvdash x = 42$. Then, the process P is executed in Q but it is precluded from execution in Q'.

Finally, the *replication* $!\,P$ means $P \parallel \mathbf{next}\,P \parallel \mathbf{next}^{\,2}P$..., 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. 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.

Extension 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 extensions to deal with reactive systems have been introduced in [23, 6, 14], non-deterministic behavior and asynchrony [14, 6], probabilistic behavior [9, 4, 15, 19], mobility [17], linearity [7], etc. We shall briefly describe some of these extensions.

Non-determinism and Asynchrony.

Being a model of reactive CCP based on the Synchronous Languages [3] (i.e. programs must be determinate and respond immediately to input signals), the tcc model is not meant for modeling nondeterministic or asynchronous temporal behavior. 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.

The ntcc calculus [14] is obtained by adding guarded-choice for modeling nondeterministic behavior and an unbounded finite-delay operator for asynchronous behavior. Computation in ntcc progresses as in tcc, except for the nondeterminism induced by the new constructs.

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

$$\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i \quad | \quad \star P$$

The guarded-choice $\sum_{i\in I} \mathbf{when}\ c_1\ \mathbf{do}\ P_i$ 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 ϵ for synchronous CCS [13] and it allows to express asynchronous behavior through the time intervals. Intuitively, process $\star P$ represents $P + \mathbf{next}\, P + \mathbf{next}^{\,2} P + ...$, i.e., an arbitrary long but finite delay for the activation of P.

Strong Pre-Emption in tcc

The work in [27] introduces the notion of strong pre-emption in 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 **unless** c **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 tcc Processes) *The* **Default** *tcc processes result from adding in the syntax in Definition 2 the following construct:*

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

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.

Stochastic and Probabilistic Behavior

When modeling a complex system, it is not longer possible to assume that enough information will be provided for all its components. Hence, models have to deal with uncertain behavior or a incomplete description of these components. A natural model is then to consider these components as a stochastic ones where a probability distribution is given to the set of outputs the 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 timed 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*

$$\textbf{new}(r, f) \textbf{ in } P$$

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

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

The authors in [19] 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 [25] 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*

$$\bigotimes_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ (P_i, a_i)$$

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

Mobile Behavior

In [17] the authors introduce *Universal Timed CCP* (`utcc`), a calculus aiming at modeling mobile reactive systems, i.e., systems than besides to interact continuously with the environment, they may change their communication structure. Here we understand mobility as in the π-calculus [20], i.e. generation and communication of private channels or links.

The basic move from `tcc` to `utcc` is to replace the ask operation **when** c **do** P by a *temporary parametric ask* constructor of the form $(\mathbf{abs}\ \vec{x}; c)\, P$. 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, $(\mathbf{abs}\ \vec{x}; c)\, P$ 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 $(\mathbf{local}\, \vec{x}; c)\, P$ can be seen as the local variables of P while $\vec{x}$ in $(\mathbf{abs}\ \vec{x}; c)\, P$ as the formal parameters of P.

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

$$(\mathbf{abs}\ \vec{x}; c)\, P$$

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

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

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 computing agents and logic formulae (see e.g., [24, 14, 5, 16]), 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*

$$A, B, \ldots := c \mid A \dot{\Rightarrow} A \mid \dot{\neg} A \mid \dot{\exists}_x A \mid \circ A \mid \Box A \mid \Diamond A$$

Here c denotes an arbitrary constraint which we shall refer to as *atomic proposition*. The intended meaning of the other symbols is the following: $\dot{\Rightarrow}$, $\dot{\neg}$ and $\dot{\exists}$ 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 $\circ$, $\Box$, and $\Diamond$ denote the temporal operators *next*, *always* and *sometime*. Intuitively $\circ\, A$, $\Diamond\, A$ and $\Box\, A$ means that the property A must hold next, eventually and always, respectively.

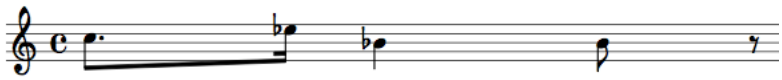
Let P be a process and A a LTL formulae specifying a given temporal property. One may wonder whether the process P satisfies the specification A, written as $P \models A$. The intended meaning of this assertion is that, regardless the input, every output of P satisfies the temporal formula A.

Let us give some examples. Since in every infinite sequence output by $\star\,\mathbf{tell}(c)$ on arbitrary inputs there must be an element entailing c, we have $\star\,\mathbf{tell}(c) \models \Diamond c$. Analogously, $!\,\mathbf{tell}(c) \models \Box c$ since all output of P must entail c. Let $P = \mathbf{tell}(c) + \mathbf{tell}(d)$. We have $P \models (c \,\dot{\vee}\, d)$ as every constraint e output by P entails either c or d. Finally, **when** c **do next** $\mathbf{tell}(d) \models c \dot{\Rightarrow} \circ d$ 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

$$P \stackrel{\texttt{def}}{=} \mathbf{tell}(note_1 = 72) \\ \parallel \mathbf{next}^{\,3}(\mathbf{tell}(note_2 = 75) \parallel \mathbf{next}\,(\mathbf{tell}(note_3 = 70) \parallel \\ \mathbf{next}^{\,4}\mathbf{tell}(note_4 = 70)))$$

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:

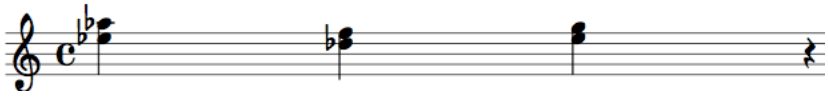
$$P_1 = \mathbf{when}\ signal = 0\ \mathbf{do}\ \mathbf{tell}(note_1 = 72) \parallel \mathbf{unless}\ signal = 0\ \mathbf{next}\ P_1$$
$$P_2 = \mathbf{when}\ signal = 3\ \mathbf{do}\ \mathbf{tell}(note_2 = 75) \parallel \mathbf{unless}\ signal = 3\ \mathbf{next}\ P_2$$
$$P_3 = \mathbf{when}\ signal = 4\ \mathbf{do}\ \mathbf{tell}(note_3 = 70) \parallel \mathbf{unless}\ signal = 4\ \mathbf{next}\ P_3$$
$$P_4 = \mathbf{when}\ signal = 8\ \mathbf{do}\ \mathbf{tell}(note_4 = 70) \parallel \mathbf{unless}\ signal = 8\ \mathbf{next}\ P_4$$

The score is then defined by a process Q

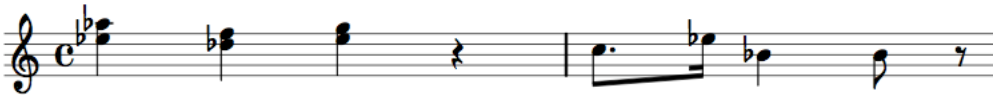
$$Q = P_1 \parallel P_2 \parallel P_3 \parallel P_4$$

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 [21]) or from a `ntcc` process modeling the conductor.

Let us now consider each of the above musical process with respect to some simple temporal operations. Suppose corresponding `ntcc` definitions for process R modeling



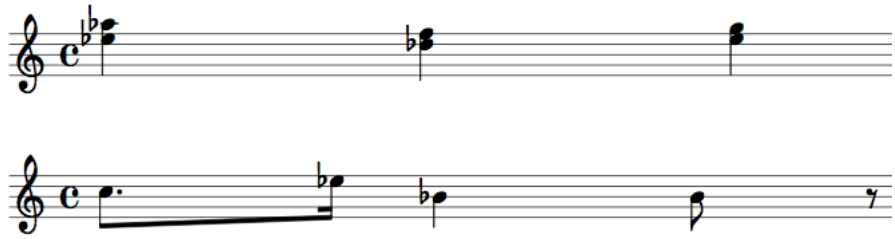
Now, we would like to model



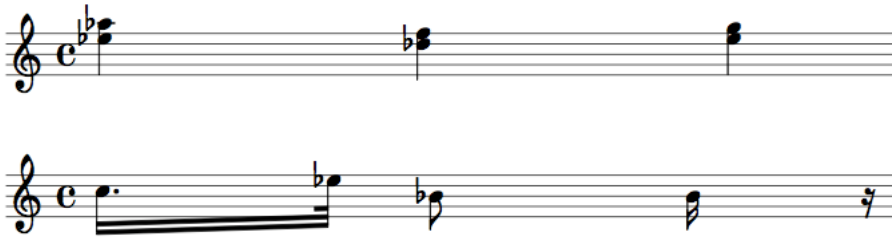
In the first representation this is done by process

$$R \parallel \mathbf{next}^{\,16} P$$

whereas in the second the conductor would have to use two hands, providing $signal_1$ for R and $signal$ 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 $signal_1$.

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

combinator	meaning
$P \;;_k\; Q$	P followed by Q delayed k units
$P \ll k$	process P stretched proportionally k units
$P \gg k$	process P expanded proportionally k units
$P \uplus Q$	process Q time-intercalated with P

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 metrics 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, 18]. We can regard music performance and composition as a complex task of defining and controlling interaction among concurrent activities. In [22], *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 examples, in particular those involving time and synchronization, are difficult to express. In this section we model two of those examples.

In the following examples we shall use the derived operators $\star_{[m,n]}P$, $_{[m,n]}P$ and $W_{(c,P)}$. The process $\star_{[m,n]}P$ 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. The construct $W_{(c,P)}$ waits until c holds and then executes P:

$$W_{(c,P)} \stackrel{\texttt{def}}{=} \textbf{when } c \textbf{ do } P \parallel \textbf{ unless } c \textbf{ next } W_{(c,P)}.$$

We shall use the more readable notation **wait** c **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.

$$x\!:\!(z) \quad \stackrel{\texttt{def}}{=} \quad \textbf{tell}(x = z) \parallel \textbf{unless } \texttt{change}(x) \textbf{ next } \; x\!:\!(z)$$

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., $\texttt{change}(x)$). Let $\text{exch}_g[x, y]$ be defined as

$$\begin{array}{rcl} \text{exch}_g[x,y] & \stackrel{\texttt{def}}{=} & \sum_v \textbf{when } x = v \textbf{ do } (\;\; \textbf{tell}(\texttt{change}(x)) \parallel \textbf{tell}(\texttt{change}(y)) \\ & & \qquad\qquad \parallel \textbf{next} \, (\; x\!:\!(g(v)) \parallel y\!:\!(v) \;) \;\;) \end{array}$$

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 $\texttt{sil} \in \mathcal{D}$ 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 ($start_i = 1$), the $i-$th musician chooses a permutation (j, k, l) of his given pattern p_i. Then, M_i spawns a process $Play^i_{(j,k,l)}$, 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 $c_i[note_i]$ specifies some value for $note_i$ different from $\texttt{sil}$. After playing his block, the $i-$th musician signals termination by setting cell $flag_i$ to 1. Furthermore, upon receiving the $go = 1$ signal, the $i-$th musician eventually starts a new block no later than $\texttt{pdur}$ which is a constant representing the sum of the durations of all patterns.

$$
\begin{aligned}
M_i \quad &\stackrel{\texttt{def}}{=} \quad !\,\mathbf{when}\ (start_i = 1)\ \mathbf{do} \\
&\qquad \sum_{(j,k,l)\in perm(p_i)} (\ Play^i_{(j,k,l)} \ \|\ \mathbf{next}^{\,j+k+l}(\ flag_i := 1 \ \| \\
&\qquad\qquad \mathbf{wait}\ (go = 1)\ \mathbf{do} \\
&\qquad\qquad\quad \star_{[0,\texttt{pdur}]}\mathbf{tell}(start_i = 1))\ )
\end{aligned}
$$

$$
\begin{aligned}
Play^i_{(j,k,l)} \quad &\stackrel{\texttt{def}}{=} \quad !_{[0,\ j-1]}\mathbf{tell}(note_i = \texttt{sil}) \ \|\ \mathbf{next}^{\,j}\mathbf{tell}(c_i[note_i]) \\
&\qquad \| \ !_{[j+1,\ j+k-1]}\mathbf{tell}(note_i = \texttt{sil}) \ \|\ \mathbf{next}^{\,j+k}\mathbf{tell}(c_i[note_i]) \\
&\qquad \| \ !_{[j+k+1,\ j+k+l-1]}\mathbf{tell}(note_i = \texttt{sil}) \ \|\ \mathbf{next}^{\,j+k+l}\mathbf{tell}(c_i[note_i])
\end{aligned}
$$

The *Conductor* process is always checking (listening) whether all the musicians play a note exactly at the same time $\bigwedge_{i\in[1,m]}(note_i \neq \texttt{sil})$. If this happens it sets the cell *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 $go = 1$ to all musician to start a new block, unless all of them have output a note at the same time (i.e., $stop = 1$).

$$
\begin{aligned}
Conductor \quad &\stackrel{\texttt{def}}{=} \quad \mathbf{wait} \bigwedge_{i\in[1,m]} (note_i \neq \texttt{sil})\ \mathbf{do}\ stop := 1 \\
&\qquad \| \,!\mathbf{when} \bigwedge_{i\in[1,m]} (flag_i = 1) \wedge (stop = 0)\ \mathbf{do}\ \ (\mathbf{tell}(go = 1) \,\|\, \prod_{i\in[1,m]} flag_i \ :=\ 0)
\end{aligned}
$$

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

$$
\begin{aligned}
Init \quad &\stackrel{\texttt{def}}{=} \quad \textstyle\prod_{i\in[1,m]}(flag_i : 0 \,\|\, \mathbf{tell}(start_i = 1)) \,\|\, stop : 0 \\
System \quad &\stackrel{\texttt{def}}{=} \quad Init \ \|\ Conductor \ \|\ \textstyle\prod_{i\in[1,m]} M_i
\end{aligned}
$$

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

$$System \models \Diamond stop = 1$$

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

$$System \models \Box stop = 0$$

does not hold, i.e., there is a run of the system for which at some time unit all the notes are different from `sil`.

Rhythm Patterns

In this section we shall model synchronization of rhythm patterns in `ntcc`. Let us first define a "metronome" process.

$$M[tick, count, \delta] \stackrel{\texttt{def}}{=} !(\ \textbf{when}\ (count\ \texttt{mod}\ \delta) = 0\ \textbf{do}\ \ tick := tick + 1 \parallel count := 0 \\ +\ \textbf{when}\ (count\ \texttt{mod}\ \delta) > 0\ \textbf{do}\ \ count := count + 1)$$

One could think of $M[tick, count, \delta]$ as a process that "ticks" (by increasing *tick*) every δ time units. This process could be controlled by the acceleration process:

$$Accel[signal, \delta] \stackrel{\texttt{def}}{=} !\ \textbf{when}\ signal = 1 \wedge \delta > 0\ \textbf{do}\ \delta := \delta - 1$$

The process $Accel[signal, k]$ can "speed up the ticks of $M[tick, count, \delta]$" by decreasing δ, if some other process, which we shall refer to as $Control[signal]$, tells $signal = 1$.

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

$$R_{(s,d,e)}[tick, note] \stackrel{\texttt{def}}{=} !\ \textbf{when}\ s \geq tick \geq e \wedge (tick - start)\ \texttt{mod}\ d = 0\ \textbf{do}\ \ Pitch[note]$$

The process $R_{(s,d,e)}[tick, note]$ runs a certain $Pitch[note]$ process, which outputs some pitch on *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:

$$System \stackrel{\texttt{def}}{=} (\textbf{local}\ tick\ \delta\ count\ signal)\ (\\ Init \parallel Control[signal] \parallel M[tick, count, \delta] \parallel Accel(signal, \delta) \parallel \\ R_{(0,30,120)}[tick, \delta] \parallel R_{(0,20,120)}[tick, \delta] \parallel R_{(0,12,120)}[tick, \delta]\ )$$

where $Init = tick : 0 \parallel \delta : 30 \parallel count : 0$.

Since the *Rhythm* processes depend on variables *tick* and δ, 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 `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

[1] M. Balaban and C. Samoun. Hierarchy, time and inheritance in music modelling. *Languages of Design*, 1(3):147–172, 1993.

[2] K. Barbar, A. Beurivé, and M. 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.

[3] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[4] Luca Bortolussi and Alberto Policriti. Modeling biological systems in stochastic concurrent constraint programming. *Constraints*, 13(1-2), 2008.

[5] F. S. de Boer, M. Gabbrielli, E. Marchiori, and C. Palamidessi. Proving concurrent constraint programs correct. *ACM Transactions on Programming Languages and Systems*, 19(5), 1997.

[6] Frank S. de Boer, Maurizio Gabbrielli, and Maria Chiara Meo. A timed concurrent constraint language. *Inf. Comput.*, 161(1), 2000.

[7] Francois Fages, Paul Ruet, and Sylvain Soliman. Linear concurrent constraint programming: Operational and phase semantics. *Information and Computation*, 165, 2001.

[8] Vineet Gupta, Radha Jagadeesan, and Prakash Panangaden. Stochastic processes as concurrent constraint programs. In *POPL '99*, New York, NY, USA, 1999. ACM.

[9] Vineet Gupta, Radha Jagadeesan, and Vijay A. Saraswat. Probabilistic concurrent constraint programming. In *Proc. of CONCUR 97*, London, UK, 1997. Springer-Verlag.

[10] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. *Formal Asp. Comput.*, 6(5):512–535, 1994.

[11] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. Design, implementation, and evaluation of the constraint language cc(fd). *J. Log. Program.*, 37(1-3), 1998.

[12] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.

[13] R. Milner. A finite delay operator in synchronous ccs. Technical Report CSR-116-82, University of Edinburgh, 1992.

[14] M. Nielsen, C. Palamidessi, and F.D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(1), 2002.

[15] Carlos Olarte and Camilo Rueda. A stochastic concurrent constraint based framework to model and verify biological systems. *CLEI electronic journal*, 9(2), 2008.

[16] Carlos Olarte and Frank D. Valencia. The expressivity of universal timed CCP: Undecidability of monadic FLTL and closure operators for security. In *Proc. of PPDP 08*, 2008.

[17] Carlos Olarte and Frank D. Valencia. Universal concurrent constraint programing: Symbolic semantics and applications to security. In *Proc. of SAC 2008*. ACM, 2008.

[18] 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.

[19] Jorge A. Pérez and Camilo Rueda. Non-determinism and Probabilities in Timed Concurrent Constraint Programming. In *Proc. of ICLP 2008*, volume 5366 of *LNCS*. Springer, 2008. Extended version available at `http://www.japerez.phipages.com`.

[20] J. Parrow R. Milner and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100, September 1992.

[21] M. Puckette. Max at seventeen. *Computer Music Journal*, 26(4): 31–43, 2002.

[22] C. Rueda, G. Alvarez, L. Quesada, G. Tamura, F. Valencia, J. Diaz, and G. Assayag. Integrating constraints and concurrent objects in musical applications: A calculus and its visual language. *Constraints*, 6(1), 2001.

[23] Vijay Saraswat, Radha Jagadeesan, and Vineet Gupta. Foundations of timed concurrent constraint programming. In *Proc. of LICS'94*. IEEE CS, 1994.

[24] Vijay A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.

[25] Roberto Segala. Modeling and verification of randomized distributed real-time systems. PhD thesis, MIT, 1995.

[26] Gert Smolka. A foundation for higher-order concurrent constraint programming. In *In Proc. of CCL 94*, 1994.

[27] V.A.Saraswat, R.Jagadeesan, and V.Gupta. Timed default concurrent constraint programming. *Journal of Symbolic Computation*, 22(5–6), 1996.