A TEMPORAL CONCURRENT CONSTRAINT CALCULUS AS AN AUDIO PROCESSING FRAMEWORK

Camilo Rueda
Universidad Javeriana-Cali
crueda@atlas.puj.edu.co

Frank Valencia
Ecole Polytechnique
frank.valencia@lix.polytechnique.fr

ABSTRACT

Audio processing systems involve complex interactions of concurrent processes. These are usually implemented using domain specific visual languages and tools more concerned with providing practical solutions than with giving formal meaning to the supplied audio unit combinators. Concurrent constraint process calculi have proved to be effective in modeling with precision a wide variety of concurrent systems. We propose using `ntcc` , a non deterministic temporal concurrent constraint calculus, to model audio processing systems. We show how the concurrent constraint nature of the calculus greatly simplify specifying complex synchronization patterns. We illustrate `ntcc` as audio processing framework by modeling unit combinators and using them ina an audio processing example.

1. INTRODUCTION

Concurrent constraint (CC) process calculi ([7]) provide formal grounds to the integration of concurrency and constraints so that non trivial properties of concurrent systems can be expressed and proved. Concurrent processes occurring in music exhibit a rich variety of synchronization schemes, calling into play different degrees of precision (i.e. partial information) about temporal or harmonic relations involving them. Musical processes span a wide range of abstraction levels, from the sound phenomena up to rhythmic or harmonic structures. The complexity of musical phenomena poses a great challenge to any computational formalism. We think that a suitable CC process calculus should provide a convenient framework to get insights into the right models to cope with this challenge, at each abstraction level. We thus borrow concepts and techniques from concurrent processes modeling to define suitable computational calculi and analyze their behavior in real settings.

In a previous work ([5]) we proposed using `ntcc` ([3]), a temporal non deterministic concurrent constraint calculus, for modeling music improvisation processes. We showed how this modeling framework gave us the possibility of easily verifying of them interesting musical properties. In this paper we are interested in pushing the idea of `ntcc` as a convenient framework for modeling concurrent processes at a lower level: that of the sound itself.

Our approach is close in spirit to [8] in looking for a minimum number of constructs allowing to conveniently express a variety of actual synchronization patterns occurring in audio processing. Also closely related to our proposal is the *Faust* system [2]. Faust is a visual audio processing language whose underlying semantics is based on an algebra of block diagrams. By providing a formal semantic base, the constructs of Faust can be given precise meaning.

Our aim is different from these two approaches. Rather than defining a full fledged domain specific language for audio processing we want to explore the pertinence of `ntcc` as a runnable specification of audio processing systems. What we gain from this approach is twofold. One the one hand, we are able to ground the development of audio processing tools on a very precise formal foundation and by this means proposing coherent higher level audio structures and operations. On the other hand, our model can give us clues for constructing formal proofs of interesting properties of a given audio process. Audio specifications in `ntcc` can be directly executed (a compiler plus an abstract machine has been implemented), thus providing a framework in which the behavior of formally specified systems can be directly observed.

We thus propose using `ntcc` as a formal base to model audio processing systems in such a way that their temporal properties can be formally proved.

The ntcc calculus inherits ideas from the tcc model [6], 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 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 a control signal s *within* the next t time units" or "the three processes must eventually output the same sample but *there is no bound* in the occurrence 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 being able to specify 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 interactive audio applications. These systems often need non-determinism and asynchrony to be modeled faithfully. To our knowledge, existing audio processing languages do not consider non determinism explicitly.

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.

In this essay we are interested in showing how non trivial audio processes calling into action different forms of partial information can be modeled in `ntcc` . We are interested in modeling audio synchronization patterns resulting from the controlled interaction of independent audio agents, such as is the case in several forms of sound synthesis. In this type of systems, state changes in one process following particular local laws must be “partially” synchronized with state changes in other processes. This poses difficulties to CCP models not including state change constructs. We show that audio processes with non determinism, partial information and state change synchronization are naturally expressed in `ntcc` . We claim this is a clear advantage of the ntcc calculus.

We also investigate ways in which properties of audio process can be formally proved. We are able to do this thanks to the logical nature of ntcc, which comes to the surface when we consider its relation with linear temporal logic: All the operators of ntcc correspond to temporal logic constructs.

We certainly do not claim `ntcc` to be an audio processing *language*. Many practical issues that are fundamental to audio processing applications are not addressed here. In particular, we believe that research on constraint systems devised specifically for audio processing is needed to achieve the level of performance required in audio processing. A clean integration of existing powerful audio processing libraries into this constraint system must be implemented before real applications can be considered.

The main contributions of this paper are: 1) to show how the expressiveness of the `ntcc` model allows simple descriptions of complex systems of interacting audio processes, 2) describing a framework for audio processing that is also capable of modeling higher level musical structures, thus giving coherence to relationships between processes at different hierarchical levels and 3) arguing that by modeling an audio process in `ntcc` one inherits a well defined logical inference system (see [3]) that can be used to prove its temporal properties.

2. THE CALCULUS

In this section we present 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.

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.

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

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

$$\sum_{i\in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i,$$

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 $\sum_{i\in I} \mathbf{when}\,(\mathtt{true})\,\mathbf{do}\,P_i$. We use $\mathbf{skip}$ 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 $\mathbf{local}\ x\ \mathbf{in}\ P$ 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. We abbreviate $\mathbf{local}\ x\ \mathbf{in}\ \mathbf{local}\ y\ \mathbf{in}\ P$ as $\mathbf{local}\ x, y\ \mathbf{in}\ P$

The process $\mathbf{next}\, P$ represents the activation of P in the next time interval. Hence, a move of $\mathbf{next}\, P$ is a unit-delay of P. The process

$$\mathbf{unless}\, c\, \mathbf{next}\, P$$

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 $\mathbf{next}$ 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. unbounded 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.

We shall use $!_I P$ and $\star_I P$, where I is an interval of natural numbers, as an abbreviation for processes $\prod_{i\in I}\mathbf{next}^i P$ and $\sum_{i\in I}\mathbf{next}^i P$, respectively. For instance, $\star_{[m,n]}P$ means that process 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 currently available 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. The formal definition (see [3] for details) introduces two reduction relations, 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 from the environment reduces, in one time unit, to Q and outputs d to the environment ". Such an observable transition is defined in terms of a sequence of internal transitions $\langle P, c\rangle \longrightarrow^* \langle Q', d\rangle$ starting in P with store c and ending in some process Q' with store d. Crudely speaking, to obtain Q we should remove from Q' what was meant to be executed only in the current time interval. Since Q is to be executed in the next time interval we should also "unfold" the sub-terms within $\mathbf{next}\,R$ expressions in Q'. As in tcc, the store does not transfer automatically from one interval to another.

To illustrate reductions in `ntcc` , consider an audio process, say $!\,P$, that continually outputs either a sample value of v or w until another process (modeling a button) Q signals the end. Process $!P \parallel Q$, for P and Q as defined below, models the example.

$$\begin{array}{rcl} P & \stackrel{\text{def}}{=} & \mathbf{when}\,(Go=1)\,\mathbf{do}\,(\mathbf{tell}\,(sample=v) \\ & & +\,\mathbf{tell}\,(sample=w)\,) \\ & & \parallel\ \mathbf{unless}\,End=1\,\mathbf{next}\,\mathbf{tell}\,(Go=1) \\ Q & \stackrel{\text{def}}{=} & \mathbf{tell}\,(Go=1)\ \parallel \star\mathbf{tell}\,(End=1) \end{array}$$

Then there is a sequence of internal transitions

$$\begin{array}{l} \langle !\,P, Go=1\rangle \longrightarrow \langle P \parallel \mathbf{next}\,!\,P, Go=1\rangle \\ \longrightarrow^* \langle \mathbf{tell}\,(sample=w) \parallel \mathbf{next}\,!\,P, Go=1\rangle \\ \longrightarrow \langle \mathbf{next}\,!\,P, sample=w \wedge Go=1\rangle \not\longrightarrow \ \ldots \end{array}$$

Initially the store contains constraint $Go = 1$ (which, as described below, will be added to the store by Q). Replicated process $!P$ then creates a copy of P and schedules itself for the next time unit. Process P ouputs w (the store gets $Go = 1 \wedge sample = w$). No further reductions are possible in the current time unit. Two processes, $!P$ and $\mathbf{tell}\ Go = 1$ are scheduled for the next time unit. So, in the case $P \parallel Q$, for an arbitrary (number of time units) $n > 1$, the following are possible transitions:

$$\begin{array}{l} \langle !P \parallel Q, true\rangle \longrightarrow^* \\ \langle \mathbf{next}\ !P \parallel \mathbf{next}\ \mathbf{tell}\ Go=1 \\ \parallel \mathbf{next}^n\mathbf{tell}(End=1), Go=1 \wedge sample=v\rangle \end{array}$$

and

$$\begin{array}{l} !P \parallel Q \xRightarrow{(\mathbf{true},\, Go=1\wedge sample=v)} \\ \quad !P \parallel \mathbf{tell}\ Go=1 \parallel \mathbf{next}^{n-1}\mathbf{tell}(End=1). \end{array}$$

The first one is the internal transition relation, whereas the second is the observable transition. Thus $!P$ continually outputs either sample v or w for an arbitrary number n of time units until the constraint $End = 1$ is put in the store.

In the examples below we use *process definition* of the form

$$\mathbf{A}(x) \stackrel{def}{\equiv} P_x$$

where P_x is a process using a variable x. A "call" of the form $A(c)$ would then launch process P_x once the variable x is substituted by c. Process definitions do not add functionality to `ntcc` since they can be defined in terms of the standard `ntcc` constructs.

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 [3]. In this section we summarize these results.

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 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 temporal operators *next*, *always* and *sometime*. Given a property A (e.g. $x > 10$) the intended meaning of $\circ A$, $\Box A$ and $\Diamond A$ is that the property holds, in the next time unit, always and eventually, respectively. We use $A \vee B$ as an abbreviation of $\neg A \Rightarrow B$ and $A \wedge B$ as an abbreviation of $\neg(\neg A \vee \neg B)$.

We shall say that process P *satisfies* A iff every infinite sequence that P can possibly output satisfies the property expressed by A. A relatively complete proof system for assertions $P \vdash A$, whose intended meaning is that P satisfies A, can be found in [3]. We shall write $P \vdash A$ if there is a derivation of $P \vdash A$ in this system.

The following notion is useful for expressing properties of processes.

Definition 3.1 (Strongest Derivable Formulae) *A formula A is the strongest temporal formula derivable for P if $P \vdash A$ and for all A' such that $P \vdash A'$, we have $A \Rightarrow A'$.*

Note that the strongest temporal formula of a process P is unique modulo logical equivalence. We give now a constructive definition of such formula.

Definition 3.2 (Strongest Formula Function) *Let the function $stf : Proc \to \mathcal{A}$ be defined as follows:*

$$\begin{array}{lcl}
stf(\mathbf{tell}(c)) & = & c \\
stf(WHEN(c_i, P_i)) & = & \left(\bigvee_{i\in I} c_i \wedge stf(P_i)\right) \\
 & & \vee \bigwedge_{i\in I} \neg c_i \\
stf(P \parallel Q) & = & stf(P) \wedge stf(Q) \\
stf(\mathbf{local}\, x\, P) & = & \exists_x stf(P) \\
stf(\mathbf{next}\ P) & = & \circ\, stf(P) \\
stf(\mathbf{unless}\ c\ \mathbf{next}\ P) & = & c \vee \circ stf(P) \\
stf(!\, P) & = & \Box\, stf(P) \\
stf(\star\, P) & = & \Diamond\, stf(P).
\end{array}$$

where the expression $WHEN(c_i, P_i))$ represents process $\sum_{i\in I} \mathbf{when}\, (c_i)\, \mathbf{do}\, P_i$.

From [3] it follows that $P \vdash stf(P)$. From this we have:

Proposition 3.3 *For every process P, $stf(P)$ is the strongest temporal formula derivable for P.*

Note that to prove that $P \vdash A$ is sufficient to prove that $stf(P) \Rightarrow A$. In addition, the proof system described in [3] gives extra mechanisms for carrying out proofs of process properties.

The idea is then to "translate" the model of an audio system in ntcc to its associated strongest temporal formula. Any temporal property one might one to prove of an audio system could then be verified in the temporal logic, even automatically, using the proof system.

4. AUDIO PROCESSING IN NTCC

The temporal nature of ntcc relates to the processing of audio samples in a natural way. Each time unit of ntcc can be associated to processing (generating or transforming) the current sample of a sequential stream. All reductions performed in a given time unit thus represent, in a way, "real time" transformations. Once a process is done computing it can reschedule itself at an appropriate time in the future, remain active for a certain time interval or simply disappear. Combinations of these possibilities provide a rich set of patterns of temporal processing.

4.1. Time and duration

Different time rates can be naturally modeled using the *next* construct. In the following example, process A is twice as fast as process B:

$$\begin{array}{l}
\mathbf{A} \overset{def}{\equiv} P_1 \,||\, \mathbf{next}\ (A) \\
\mathbf{B} \overset{def}{\equiv} P_2 \,||\, \mathbf{next}\ ^2(B)
\end{array}$$

The notion of duration can be expressed as pairs of *on/off* events, $P_{on} \,||\, \mathbf{next}\ ^d P_{off}$, or explicitly as a process executing itself during a certain time range: $!_{[i,j]} P$. Processes "beating" time at different rates can be used by other processes to express a variety of synchronization schemes:

$$\begin{array}{l}
\mathbf{TICK_1(i)} \overset{def}{\equiv} \mathbf{tell}\,(beat_1 = i) \,||\, \mathbf{next}\ ^2(TICK_1(i+1)) \\
\mathbf{TICK_2(i)} \overset{def}{\equiv} \mathbf{tell}\,(beat_2 = i) \,||\, \mathbf{next}\ ^3(TICK_2(i+1))
\end{array}$$

$$\begin{array}{l}
\mathbf{TICK_1(0)} \,||\, \mathbf{TICK_2(2)} \\
||\, \mathbf{!when}\, beat_2 \neq beat_1\ \mathbf{do\ next}\ P_1 \\
||\, \mathbf{!unless}\, beat_2 \neq beat_1\ \mathbf{next}\ P_2
\end{array}$$

Process P_1 will be executed at times $2, 8, 20, 26, 32, 38, \ldots$ whereas process P_2 will be run at all times except those. Notice how being able to reason on absence of information (the *unless* construct) allows expressing this behavior in a simple way. As can be seen in the above example, time in ntcc can be regarded as "advanced" by the user (much like in *ChucK*, see [8]) or it can be seen as controlled by a sampling process providing a new sample at each ntcc time slot. In both cases time is independent from the underlying hardware (real) timing. This fact greatly simplifies proving temporal properties of processes.

In addition, the blocking behavior of constraints under lack of sufficient information provides a very natural and expressive synchronizing mechanism. This feature gives a declarative flavor to concurrency that avoids in most cases dealing with complex synchronizing schemes such as semaphores or monitors.

4.2. Unit processing elements

Unit generators can be represented as processes taking an input stream and performing some given transformation on it. The input and output streams are modeled by suitable global variables accessible to each process. The environment (or a process) sets the value of the input variable at each time unit. Interacting processes cooperate to compute a value for the variable representing the output stream (i.e. they assert constraints on that variable). It is important to remark that all variables are *logical*. Their value cannot be changed during the same time unit. A unit generator is of the form:

$$\mathbf{SOUND\_UNIT_g(in, out)} \overset{def}{\equiv} !\mathbf{tell}\ (out = g(in))$$

An envelope process, for example, can be defined as:

$$\begin{aligned} &\mathbf{ENVEL_f(in)} \overset{def}{\equiv} \\ &\quad \mathbf{tell}\ (out = in) \\ &\quad \|\ \mathbf{next}\ (ENVEL(f(in))) \end{aligned}$$

Applying the envelope to a unit generator is achieved by the parallel composition of the above:

$$\begin{aligned} &\mathbf{local}\ x, y, out\ \mathbf{in} \\ &\quad SOUND\_UNIT_g(in, x) \,\|\, ENVEL_f(in, y) \\ &\quad \|\ !\mathbf{tell}\ (out = x \times y) \end{aligned}$$

In fact, a wide variety of combinations among processing elements is easily defined. Consider, for example, those proposed in the algebra of block diagrams of [2]:

Given two audio processes A and B a process S running them sequentially can be defined as

$$\begin{aligned} &\mathbf{A}(in_1, in_2, out) \overset{def}{\equiv} P_1 \\ &\mathbf{B}(in_1, in_2, out) \overset{def}{\equiv} P_2 \\ &\mathbf{S}(in_1, in_2, in_3, out) \overset{def}{\equiv} \\ &\quad \mathbf{local}\ z\ \mathbf{in}\ (A(in_1, in_2, z) \,\|\, B(in_3, z, out)) \end{aligned}$$

By using local variables all sorts of signal splitting or merging between sequential processes can be defined. Say process C taking three input streams and producing one output stream is to be run after processes A and B. Two of the inputs of Process C come from splitting the output of A and the third one from adding the outputs of A and B. This can be defined as follows (process S_2, see figure 1 (a)):

$$\begin{aligned} &\mathbf{C}(in_1, in_2, in_3, out) \overset{def}{\equiv} P_3 \\ &\mathbf{S_2}(in_1, in_2, in_3, in_4, out) \overset{def}{\equiv} \\ &\quad \mathbf{local}\ x, y, z\ \mathbf{in} \\ &\qquad (A(in_1, in_2, x) \,\|\, B(in_3, in_4, y) \,\| \\ &\qquad \mathbf{tell}\ (z = x + y) \,\|\, C(x, x, z, out)) \end{aligned}$$

Since it corresponds to a primitive of the calculus, parallel composition of stream processing units (process D below) is straightforward:

$$\begin{aligned} &\mathbf{D}(in_1, in_2, in_3, in_4, out_1, out_2) \overset{def}{\equiv} \\ &\quad A(in_1, in_2, out_1) \,\|\, B(in_3, in_4, out_1, out_2) \end{aligned}$$

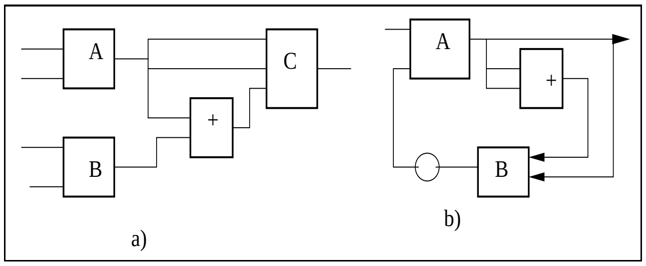

Figure 1. a) Merging/splitting and b) loopback stream processing units

Notice that in all examples above only the parallel composition operator is used for combining processes. The right synchronization is nevertheless guaranteed because of the *implicit blocking ask* of constraints. When no information on the value of a variable is available (e.g.. a sample has not been computed yet) constraints needing that value simply block.

Processes using loopback signals are also straightforward (see figure 1 (b)).

$$\begin{aligned} &\mathbf{E}(i) \overset{def}{\equiv} \\ &\quad \mathbf{local}\ z, w\ \mathbf{in} \\ &\qquad (A(i, in_2, out) \,\|\, \mathbf{tell}\ (w = out + out) \,\| \\ &\qquad B(out, w, z) \,\|\, \mathbf{next}\ (E(z))) \end{aligned}$$

Constructs dealing explicitly with time units can be used to model a variety of control mechanisms at the sample level, much in the same way as explained above for time rates. For example, a process P that is to be executing from sample m to sample n is readily defined by

$$\mathbf{A}(in_1, in_2, , out) \overset{def}{\equiv} !_{[m,n]}\, P.$$

Notice how this avoids defining control processes dealing with boolean "signals". The same goes for guarded eventuality where a process is launched at an unspecified time within a given range, $\mathbf{A}(in_1, in_2, out) \overset{def}{\equiv} \ *_{[m,n]}\, P.$

Conditional execution is given by guarded commands. A process transforming the maximum of two signals can be given by,

$$\begin{aligned} &\mathbf{C} \overset{def}{\equiv} \\ &\quad !(\mathbf{when}\ in_1 \geq in_2\ \mathbf{do}\ \mathbf{tell}\ (out = in_1 * in_1) \\ &\quad + \mathbf{when}\ in_1 < in_2\ \mathbf{do}\ \mathbf{tell}\ (out = in_2 * in_2)) \end{aligned}$$

The type of conditionals is only limited by the given constraint system. In fact, a process might compute partial information on a signal that can be used for synchronization purposes. For example, process C above could have been defined as

$$\begin{aligned} &\mathbf{C}(in_1, in_2, out) \overset{def}{\equiv} \\ &\quad !(\mathbf{when}\ in_1 \geq in_2\ \mathbf{do}\ \mathbf{tell}\ (out > in_1 * in_1) \\ &\quad + \mathbf{when}\ in_1 < in_2\ \mathbf{do}\ \mathbf{tell}\ (out > in_2 * in_2)) \end{aligned}$$

where the output is not totally determined. This could be used to control some other process, as in

$$\mathbf{D}(in_1, in_1) \overset{def}{\equiv} !(\mathbf{unless}\, in_2 > th\, \mathbf{next}\, out = in_2 {+} in_1)$$

Processes C and D are synchronized in

$$\mathbf{local}\, x\, \mathbf{in}\, (C(z, w, x) \,||\, D(i, x, j))$$

4.3. Buffering

There are several ways of representing buffering (windowing) and serializing processes. They could be modeled by processes that schedule themselves at the appropriate time in the future. For example, consider a buffer of size k that is to be filled with consecutive samples and some process P_{buf} that performs some computation on the buffer once it is filled with data. Furthermore, suppose that at each time unit some process feeds the value of a sample in a global variable $sample$. The $Merger$ agent below schedules a constraint setting each sample value into a position of the buffer at the right time, Process P_{buff} is launched by agent $Apply$ exactly when all buffer values have been set:

$$\begin{aligned}&\mathbf{Merger}_k(i) \overset{def}{\equiv}\\ &\quad \mathbf{next}^{\,k-i\%k}(\mathbf{tell}\,(buf_i = in))\\ &\quad ||\, \mathbf{next}\, Merger_k(i+1)\end{aligned}$$

$$\mathbf{Apply}_k \overset{def}{\equiv} \;\mathbf{next}^{\,k}(P_{buf} \,||\, Apply_k)$$

where the symbol $\%$ denotes the arithmetic modulo operation. The opposite of $Merge$ is a serialization process that takes all buffer values and feed them one by one at consecutive time units:

$$\begin{aligned}&\mathbf{Serialize}_k \overset{def}{\equiv}\\ &\textstyle\prod_{i\in 1..k} \sum_v \mathbf{when}\, buf_i = v\, \mathbf{do}\, \mathbf{next}^{\,i}\mathbf{tell}\,(sample = v)\end{aligned}$$

where $\prod_{i\in 1..k} P_i$ is a shorthand for $P_1 \,||\, \ldots \,||\, P_k$.

A more natural way of representing a buffer is, of course, as a sort of read/write table. This tables are persistent structures that can easily be modeled in `ntcc` as a collection of *cells*. Cells are defined with the expression $x : (z)$. This defines a cell x with initial value z. Cells are updated with the *exchange* operation: *exch*$_v[x, y]$, This form assigns (in the next time unit) v to cell x and at the same time, variable y gets the current value of x. Cells do not add functionality to `ntcc` . They can be expressed by using the standard primitives (see [3]). Using cells, a read/write table of size k can be represented by the following pair of processes:

$$\begin{aligned}&\mathbf{READ}_k(index, out) \overset{def}{\equiv}\\ &\quad \textstyle\sum_{i\in 1..k} \mathbf{when}\, index = i\, \mathbf{do}\, x_i : (out)\\ &\mathbf{WRITE}_k(index, value, out) \overset{def}{\equiv}\\ &\quad \textstyle\sum_{i\in 1..k} \mathbf{when}\, index = i\, \mathbf{do}\, \mathit{exch}_{value}[x_i, y]\end{aligned}$$

We can imagine some process initializing the above read/write table with a stream of samples:

$$\begin{aligned}&\mathbf{Initialize}_k(i) \overset{def}{\equiv}\\ &\quad \mathbf{when}\, i \le k\, \mathbf{do}\, \mathbf{local}\, x\, \mathbf{in}\\ &\qquad Write_k(i, sample, x) \,||\, \mathbf{next}\, Initialize_k(i+1)\end{aligned}$$

4.4. Synchronization

In many audio streaming applications synchronization on external events is frequent. In `ntcc` the occurrence of an external event is observed by some information it adds to the current store. The presence (or absence) of this information can be readily tested by other processes so that synchronization patterns emerge in a natural manner. For example, say pushing some button b on a device (or GUI) should trigger audio processing whereas pushing button s should stop it. All processes could then synchronize on some $trig$ signal controlled by a process observing the buttons:

$$\begin{aligned}&\mathbf{TRIGGER}(i) \overset{def}{\equiv}\\ &\quad \mathbf{when}\, push_b\, \mathbf{do}\\ &\qquad \mathbf{next}\,(\mathbf{tell}\, trig = 1 \,||\, TRIGGER(1))\\ &\quad ||\, \mathbf{when}\, push_s\, \mathbf{do}\\ &\qquad \mathbf{next}\,(\mathbf{tell}\, trig = 0 \,||\, TRIGGER(0))\\ &\quad ||\, \mathbf{unless}\, push_b \vee push_s\\ &\qquad \mathbf{next}\,(\mathbf{tell}\, trig = i \,||\, TRIGGER(i))\end{aligned}$$

The environment should of course guarantee that b and s are not pushed at the same time. Each unit processing element in the system could then simply multiply its current input sample by the value of $trig$. The above agent could be launched in a expression of the form

$$\mathbf{local}\, x\, \mathbf{in}\, TRIGGER(x)...$$

In this way, all processing units would be blocked until button b is pushed. For example, a noise generator (see [2]) making part of the system can be defined as follows:

$$\begin{aligned}&\mathbf{NOISE}(v) \overset{def}{\equiv}\\ &\quad \mathbf{tell}\,(out = v \times trig \times 1/2147483647)\\ &\quad ||\, \mathbf{next}\,(NOISE(v \times 1103515245 + 12345))\end{aligned}$$

The next example, taken from [2], rounds up the illustration of audio processing using `ntcc` .

4.5. An audio processing example

In [2] a block diagram algebra is presented as a formal base for audio stream processing. We take the same example of the Karplus-strong algorithm to complete the description of `ntcc` as a framework for audio processing. Figure 2 shows the block diagram. The Delay subsystem uses a table lookup indexing. The index process is depicted in figure 3. Figure 4 shows the implementation of a delay block using a read/write table.

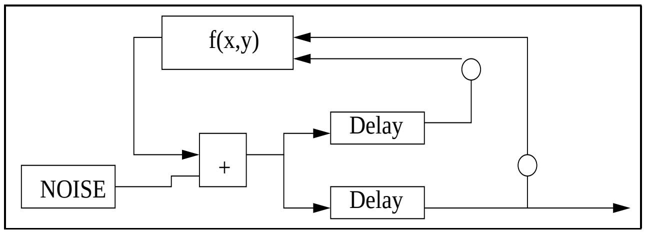

Figure 2. Karplus/strong system

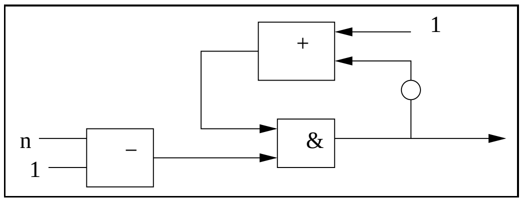

Figure 3. Table indexing

The INDEX block is modeled as

$$\begin{aligned}&\mathbf{INDEX}_n(v) \overset{def}{\equiv}\\ &\ \mathbf{tell}\ (out = v\&(n-1))\\ &\ \|\ \mathbf{next}\ (INDEX((v\&(n-1))+1))\end{aligned}$$

The representation of the delay block uses the INDEX and READ agents (the latter decorated with some extra arguments defining the offset) defined previously.

$$\begin{aligned}&\mathbf{DELAY}_{n,d}(v) \overset{def}{\equiv}\\ &\ \ \mathbf{local}\ ix, y, z, w\ \mathbf{in}\\ &\ \ \ (\mathbf{local}\ out\ \mathbf{in}\ (INDEX(v)\ \|\ !\mathbf{tell}\ (ix = out)))\\ &\ \ \ \|\ !\mathbf{tell}\ (y = ix - d)\ \|\ !\mathbf{tell}\ (z = n - 1)\\ &\ \ \ \|\ !\mathbf{tell}\ (w = y\&z)\\ &\ \ \ \|\ !READ_n(ix, input, w, out)\end{aligned}$$

Finally, the system is assembled as:

$$\begin{aligned}&\mathbf{local}\ ns, out_f\ \mathbf{in}\\ &\ \ (\mathbf{local}\ out\ \mathbf{in}\ (NOISE(0)\ \|\ !\mathbf{tell}\ (ns = out)))\\ &\ \ (\mathbf{local}\ input\ \mathbf{in}\\ &\ \ \ \ (!\mathbf{tell}\ (input = ns + out_f)\ \|\ DELAY_{n,d}(0)))\\ &\ \ \|\ LOOPB(0)\ \|TRIGGER(0)\|\ \star\ \mathbf{tell}\ push_b\end{aligned}$$

where the loopback process is

$$\begin{aligned}&\mathbf{LOOPB}(v) \overset{def}{\equiv}\\ &\ \mathbf{tell}\ (out_f = v)\\ &\ \|\ \mathbf{next}\ (LOOPB(f(out, out)))\end{aligned}$$

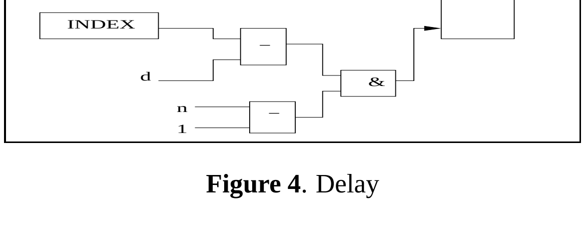

Figure 4. Delay

5. CONCLUSIONS AND FUTURE WORK

We have illustrated how temporal concurrent constraint process calculi can be suitable frameworks for specifying audio processing systems. In particular, we used the ntcc calculus to model the usual unit combinators in audio stream processing languages and showed how this modeling formalism gives compact and precise definitions of audio stream systems.

We argued that the advantage of using ntcc is that a well-defined formalism with a clear semantics is inherited. In addition, the logical nature of ntcc allows to easily express and prove temporal properties of audio processes. Since ntcc had previously shown to be a convenient formalism to model higher level musical process, coupling those processes to lower level audio operations would become much easier when the latter are also modeled in the same formalism.

We gave examples to show that in essence the user can always combine audio processes in ntcc using only parallel composition since different patterns of synchronization are easily achieved using constraint entailment.

Although in its current setting ntcc is certainly far from a practical audio processing language we think its expressiveness provide a powerful framework for modeling complex audio processing. Moreover, our research group has implemented an abstract machine for ntcc that is efficient enough for real time programming of LEGO robots (see [4]). The behavior of audio systems modeled in ntcc can thus be directly observed, at least for some processes.

The real time requirements for audio processing is, of course, a lot more demanding. The above mentioned abstract machine uses a general finite domains constraint system. For practical audio applications this might not be adequate. We plan to continue our research on efficient constraint systems specifically adapted to audio processing.

Recently (see [1]) an extension of ntcc with stochastic constructs has been defined. This can be specially interesting for music improvisation systems dealing with stochastic control of audio processes. We are currently exploring models of audio operations using this extended calculus.

6. REFERENCES

[1] C. Olarte and C. Rueda ”A stochastic Non deterministic Temporal Concurrent Constraint calculus”, *submitted to: XXV International Conference of the Chilean Computing Society*, Chile, 2005.

[2] Y. Orlarey and D. Fober and S. Letz. ”Syntactical and Semantical aspects of Faust”, *Soft Computing, A fusion of foundations, methodologies and applications*, Vol. 8 (9). Springer, 2004.

[3] C. Palamidessi and F. Valencia. ”A Temporal Concurrent Constraint Programming Calculus”, *Proc. of the Seventh International Conference on Principles and Practice of Constraint Programming*, 2001.

[4] P. Munoz and R. Hurtado. ”Programming Robotic Devices with a Timed Concurrent Constraint Calculus”, *Proceedings Principles and Practice of Constraint Programming, CP2004,* LNCS3258, Springer, 2004.

[5] C. Rueda and F. Valencia. ”Proving musical properties Using a temporal Concurrent Constraints calculus”, *Proceedings of ICMC2002*, Sweden,2002.

[6] V. Saraswat and R. Jagadeesan and V. Gupta. ”Foundations of Timed Concurrent Constraint Programming”, *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science*, 1994.

[7] V. Saraswat and M. Rinard and P. Panangaden. ”The semantic foundations of concurrent constraint programming”, *POPL’91. Proceedings of the eighteenth annual ACM symposium on Principles of programming languages*, 1991.

[8] G. Wang and P.R. Cook. ”ChucK: A Concurrent, On-the-fly Audio Programming Language”, *In Proceedings of the International Computer Music Conference (ICMC)*, Singapore, 2003.

[9] R. Milner, J. Parrow and D. Walker. ”A Calculus of Mobile Processes, Parts I and II”, *Journal of Information and Computation*,1992.