A Declarative Language for Dynamic Multimedia Interaction Systems ⋆

Carlos Olarte[1,2] and Camilo Rueda[2,3]

[1] INRIA and LIX, École Polytechnique, France
{colarte}@lix.polytechnique.fr
[2] Pontificia Universidad Javeriana Cali, Colombia
{crueda}@cic.javerianacali.edu.co
[3] IRCAM, France

Abstract. Universal Timed Concurrent Constraint Programming (utcc) is a declarative model for concurrency tied to logic. It aims at specifying mobile reactive systems, i.e., systems that continuously interact with the environment and may change their communication structure. In this paper we argue for utcc as a declarative model for dynamic multimedia interaction systems. Firstly, we show that the notion of constraints as partial information allows us to neatly define temporal relations between interactive agents or events. Secondly, we show that mobility in utcc allows for the specification of more flexible and expressive systems. Thirdly, by relying on the underlying temporal logic in utcc, we show how non-trivial temporal properties of the model can be verified. We give two compelling applications of our approach. We propose a model for dynamic interactive scores where interactive points can be defined to adapt the hierarchical structure of the score depending on the information inferred from the environment. We then broaden the interaction mechanisms available for the composer in previous (more static) models. We also model a music improvisation system based on the factor oracle that scales up to situations involving several players, learners and improvisers.

1 Introduction

Process calculi provide a language in which the structure of terms represents the structure of processes together with an operational semantics to represent computational steps. Concurrent Constraint Programming (CCP) [13] has emerged as a declarative model for concurrency tied to logic. In CCP, concurrent systems are specified by means of constraints (e.g. $x+y \geq 10$) representing partial information about certain variables. This way, agents (or processes) interact with each other by telling and asking information represented as constraints in a global store: A process $\mathbf{tell}(c)$ adds the constraint c, thus making it available to other processes. A *positive ask* $\mathbf{when}\ c\ \mathbf{do}\ P$ remains blocked until the store is strong enough to entail c; if so, it behaves like P.

⋆ This work has been partially supported by FORCES, an INRIA's *Equipe Associée* between the teams COMETE (INRIA), the Music Representation Research Group (IRCAM), and AVISPA.

Interactivity in multimedia systems has become increasingly important. The aim is to devise ways for the machine to be an active partner in a collective behavior constructed dynamically by many actors. In its simplest form, a musician signals the computer when processes should be launched or stopped. In more complex forms the machine is always actively adapting its behavior according to the information derived from the activity of the other partners. To be coherent these machine actions must be the result of a complex adaptive system composed of many agents that should be coordinated in precise ways. Constructing such systems is a challenging task. Moreover, ensuring their correctness poses a great burden to the usual test-based techniques. In this setting, CCP has much to offer: CCP calculi are explicitly designed for expressing complex coordination patterns in a very simple way by means of constraints. In addition, their declarative nature allows formally proving properties of systems modeled with them.

Interactive scores [3] are models for reactive music systems adapting their behavior to different types of intervention from a performer. Weakly defined temporal relations between components in an interactive score specifies loosely coupled music processes potentially changing their properties in reaction to stimulus from the environment (say, a performer). An interactive score defines a hierarchical structure of processes. Musical properties of a process depend on the context in which it is located. Although the hierarchical structure has been treated as static in previous works, there is no reason it should be so. A process, in reaction to a musician action, for example, could be programmed to move from one context to another or simply to disappear. Imagine, for instance, a particular set of musical materials within different contexts that should only be played when an expected information from the environment actually takes place. Modeling this kind of interactive score mobility in a coherent way is greatly simplified by using the calculus described in this paper.

Musical improvisation is another natural context for interacting agents. Improvisation is effective when agents behavior adapts to what has been learned in previous interactions. A music style-learning/improvisation scheme such as Factor Oracle (FO) [1, 5] can be seen as a reactive system where several learning and improvising agents react to information provided by the environment or by other agents. In its simplest form three concurrent agents, a player, a learner and an improviser must be synchronized. Since only three independent processes are active, coordination can be implemented without major difficulties using traditional languages and tools. The question is whether such implementations would scale up to situations involving several concurrent agents. For an implementation using traditional languages the complexity of such systems would most likely impose many simplifications in coordination patterns if behavior is to be controlled in a significant way. A CCP model, as described here, provides a compact and simple model of the agents involved in the FO improvisation, one in which coordination is automatically provided by the blocking *ask* construct of the calculus. Moreover, additional agents could easily be incorporated in the system. As an extra bonus, fundamental properties of the constructed system can be formally verified in the model.

In this paper we argue for Universal Timed CCP (`utcc`) [10] as a declarative language for the modeling and verification of multimedia interaction systems. The `utcc` calculus is a timed extension of CCP with the ability to model *mobile reactive* sys-

tem, i.e., systems that continuously interact with the environment and may change their communication structure.

After a brief introduction of utcc in Section 2, our contributions are as follows. In Section 3, we propose a utcc model for interactive scores where the interactive points allow the composer to dynamically change the hierarchical structure of the score. We then broaden the interaction mechanisms available for the user in previous (more static) models, e.g., [4], where temporal objects cannot be moved to different contexts according to the information derived from the environment. We also provide a framework based on the underlying linear temporal logic of utcc to formally verify fundamental properties of the constructed system. For instance, we can verify if certain musical structure is not played due to the absence of a stimulus from the environment. In Section 4 we model a music improvisation system based on the factor oracle that scales up to situations involving several agents and offers a more compact and efficient representation of the data structure wrt the model in [5]. Section 5 concludes the paper.

An extended version of this work, including further details, is available at [8].

2 Preliminaries

CCP-based languages are parametric in a constraint system [13] defining the kind of constraints that can be used in the program. Here, *constraints* $c, d, \ldots$ are understood as formulae in a first-order language. If the information of d can be *entailed* (or deduced) from the information represented by c we write $c \vdash d$ (e.g. $pitch > 64 \vdash pitch > 48$).

Universal timed CCP (utcc) [9] extends Timed CCP (tcc) [12] for mobile reactive systems. Time in utcc is conceptually divided into *time intervals* (or *time-units*). In a particular time-unit, a utcc process P gets an input c from the environment, it executes with this input as the initial *store*, and when it reaches its resting point, it *outputs* the resulting store d to the environment. Furthermore, the resting point determines a residual process, which is then executed in the next time interval.

Processes in utcc are built by the following syntax:

$$P, Q := \mathbf{skip} \mid \mathbf{tell}(c) \mid (\mathbf{abs}\ \boldsymbol{x}; c)\ P \mid P \parallel Q \mid (\mathbf{local}\ \boldsymbol{x}; c)\ P \mid \\ \mathbf{next}\ P \mid \mathbf{unless}\ c\ \mathbf{next}\ P \mid !\, P$$

A process $\mathbf{skip}$ represents *inaction*. A process $\mathbf{tell}(c)$ adds c to the store in the current time interval, thus making it available to other processes.

In utcc, the CCP ask operator $\mathbf{when}\ c\ \mathbf{do}\ P$ (executing P if c can be deduced) is replaced by the *abstraction* operator $(\mathbf{abs}\ \boldsymbol{x}; c)\ P$. This construct is a parameterized ask where $P[\boldsymbol{t}/\boldsymbol{x}]$ is executed for *all the terms* $\boldsymbol{t}$ s.t $c[\boldsymbol{t}/\boldsymbol{x}]$ is entailed by the store.

A process $P \parallel Q$ denotes P and Q running in parallel possibly "communicating" via the common store. The process $(\mathbf{local}\ \boldsymbol{x}; c)\ P$ behaves like P but the information c about the variables in $\boldsymbol{x}$ is local to P. We shall omit c in $(\mathbf{local}\ \boldsymbol{x}; c)\ P$ when $c \equiv \mathtt{true}$.

From a programming language perspective, $\boldsymbol{x}$ in $(\mathbf{local}\ \boldsymbol{x}; c)\ P$ can be viewed as the local variables of P while $\boldsymbol{x}$ in $(\mathbf{abs}\ \boldsymbol{x}; c)\ P$ as the formal parameters of P. This way, abstractions can encode recursive definitions of the form $X(\boldsymbol{x}) \stackrel{\mathtt{def}}{=} P$ (see [8]).

The unit-delay $\mathbf{next}\ P$ executes P in the next time interval. The (weak) time-out $\mathbf{unless}\ c\ \mathbf{next}\ P$ executes P in the next time-unit iff c *cannot* be entailed by the final

store at the current time interval. 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.

We shall also use the derived operator $(\mathbf{wait}\ \boldsymbol{x}; c)\ \mathbf{do}\ P$ that *waits*, possibly for several time-units, until for some t, $c[\boldsymbol{t}/\boldsymbol{x}]$ holds and then it executes $P[\boldsymbol{t}/\boldsymbol{x}]$ (see [10]).

An Example. The abstraction operator allows us to communicate (local) names or variables between processes, i.e., mobility in the sense of the π-calculus [7]. Let us give a simple example of this situation. Let P be a process modeling a musician playing notes at different time-units, and Q be an improvisation system which after "reading" the note played by P performs some action R. Roughly, this scenario can be modeled as follows

$$P \stackrel{\text{def}}{=} \mathbf{tell}(\mathtt{play}\,(A)) \parallel \mathbf{next}\,(\mathbf{tell}(\mathtt{play}\,(G)) \parallel \mathbf{next}\,\mathbf{tell}(\mathtt{play}\,(B))) \ldots$$
$$Q \stackrel{\text{def}}{=} !\,(\mathbf{abs}\,x; \mathtt{play}(x))\,R$$

When executing $P \parallel Q$, we observe, e.g., $R[G/x]$ in the second time-unit. This means that P and Q synchronized on the constraint $\mathtt{play}(\cdot)$ and the note played by P (i.e. G) was read by Q and then processed by R. See [8] for a more involved example defining synchronization of multiple agents.

Logic Characterization. The `utcc` calculus enjoys a declarative view of processes as first-order linear-time temporal logic (FLTL) formulae [6]. This means that processes can be seen, at the same time, as computing agents and as logic formulae.

Formulae in FLTL are built from the following syntax

$$F, G, \ldots := c \mid F \wedge G \mid \neg F \mid \exists x F \mid \circ F \mid \Box F.$$

where c is a constraint. The modalities $\circ F$ and $\Box F$ stand for resp., that F holds *next* and *always*. We use $\forall x F$ for $\neg\exists x \neg F$, and the *eventual* modality $\Diamond F$ as an abbreviation of $\neg\Box\neg F$. See [6] for further details on this logic.

Processes in `utcc` can be represented as FLTL formulae as follows:

$$\begin{array}{llllll}
[\![\mathbf{skip}]\!] & = \mathtt{true} & [\![\mathbf{tell}(c)]\!] & = c & [\![P \parallel Q]\!] & = [\![P]\!] \wedge [\![Q]\!] \\
[\![(\mathbf{abs}\,\boldsymbol{y}; c)\,P]\!] & = \forall \boldsymbol{y}(c \Rightarrow [\![P]\!]) & [\![(\mathbf{local}\,\boldsymbol{x}; c)\,P]\!] & = \exists \boldsymbol{x}(c \wedge [\![P]\!]) & & \\
[\![\mathbf{next}\,P]\!] & = \circ [\![P]\!] & [\![\mathbf{unless}\,c\,\mathbf{next}\,P]\!] & = c \vee \circ [\![P]\!] & [\![!\,P]\!] & = \Box [\![P]\!]
\end{array}$$

Let $A = [\![P]\!]$. Roughly, $A \vdash \Diamond c$ (i.e., c eventually holds in A) iff the process P eventually outputs c (see [8, 9] for further details).

3 A Model for Dynamic Interactive Scores

An interactive score [3] is a pair composed of *temporal objects* and Allen temporal relations [2]. In general, each object is comprised of a start-time, a duration, and a procedure. The first two can be partially specified by constraints, with different constraints giving rise to different types of temporal objects, so-called *events* (duration equals zero), *textures* (duration within some range), *intervals* (textures without procedures) or *control-points* (a temporal point occurring somewhere within an interval object). The procedure gives operational meaning to the action of the temporal object. It could just be playing a note or a chord, or any other action meaningful for the composer.

Figure 1, based on one from [3], shows an interactive score where temporal objects are represented as *boxes*. Objects are T_i, durations D_i. Object T_4 is a control point, whereas T_0 and T_3 are intervals. Duration D_3 should be such that $D_s \leq D_3 \leq D_f$. The whole

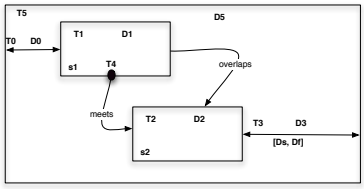


Fig. 1. Interactive score

temporal structure is determined by the hierarchy of temporal objects. Suppose that, as a result of the information obtained by the occurrence of an event, object T_2 should no longer synchronize with a control-point inside T_1 but, say, with a similar point inside T_5. This very simple interaction cannot be modeled in the standard model of interactive scores [3]. Another example is an object waiting for some interaction from the performer within some temporal interval. If the interaction does not occur, the composer might then determine to probe the environment again later when a similar musical context has been defined. This amounts to moving the waiting interval from one box to another.

The model. Figure 2 shows our model for dynamic interactive scores. The process $BoxOperations$ may perform the following actions:

- $\texttt{mkbox}(id, d)$: defines a new box with id id and duration d. The start time is defined as a new (local) variable s whose value will be constrained by the other processes.
- $\texttt{destroy}(id)$: firstly, it retrieves the box sup which contains the box id. If the box id is not currently playing, in the next time-unit, it drops the boundaries of id by inserting all the boxes contained in id into sup.
- $\texttt{before}(x, y)$: checks if x and y are contained in the same box. If so, the constraint $\texttt{bf}(x, y)$ is added.
- $\texttt{into}(x, y)$: dictates that the box x is into the box y if x is not currently playing.
- $\texttt{out}(x, y)$: takes the box x out of the box y if x is not currently playing.

Process $Constraints$ adds the necessary constraints relating the start times of each temporal object to respect the hierarchical structure of the score. For each constraint of the form $\texttt{in}(x, y)$, this process dictates that the start time of x must be less than the one of y. Furthermore, the end time of y (i.e. $d_y + s_y$) must be greater than the end time of x. The case for $\texttt{bf}(x, y)$ can be explained similarly.

The process $Persistence$ transfers the information of the hierarchy (i.e. box declarations, `in` and `bf` relations) to the next time-unit.

The process $Clock$ defines a simple clock which binds the variable t to the value v in the current time-unit and to $v + 1$ in the next time-unit.

$$
\begin{array}{rl}
BoxOperations \stackrel{\text{def}}{=} & (\textbf{abs}\ id, d; \texttt{mkbox}(id, d)) \\
& \quad (\textbf{local}\ s)\ \textbf{tell}(\texttt{box}(id, d, s)) \\
\| & (\textbf{abs}\ id; \texttt{destroy}(id)) \\
& \quad (\textbf{abs}\ x, sup; \texttt{in}(x, id) \wedge \texttt{in}(id, sup)) \\
& \quad\quad \textbf{unless}\ \texttt{play}(id)\ \textbf{next tell}(\texttt{in}(x, sup)) \\
\| & (\textbf{abs}\ x, y; \texttt{before}(x, y))\ \textbf{when}\ \exists_z(\texttt{in}(x, z) \wedge \texttt{in}(y, z))\ \textbf{do} \\
& \quad\quad \textbf{unless}\ \texttt{play}\,(y)\ \textbf{next tell}(\texttt{bf}(x, y)) \\
\| & (\textbf{abs}\ x, y; \texttt{into}(x, y))\ \textbf{unless}\ \texttt{play}\,(x)\ \textbf{next tell}(\texttt{in}(x, y)) \\
\| & (\textbf{abs}\ x, y; \texttt{out}(x, y))\ \textbf{when}\ \texttt{in}(x, y)\ \textbf{do} \\
& \quad\quad \textbf{unless}\ \texttt{play}\,(x)\ \textbf{next}\ (\textbf{abs}\ z, \texttt{in}(y, z); \textbf{tell}(\texttt{in}(x, z))) \\
Constraints \stackrel{\text{def}}{=} & (\textbf{abs}\ x, y; \texttt{in}(x, y))\ (\textbf{abs}\ d_x, s_x; \texttt{box}(x, d_x, s_x)) \\
& \quad (\textbf{abs}\ d_y, s_y; \texttt{box}(y, d_y, s_y)) \\
& \quad\quad \textbf{tell}(s_y \leq s_x) \| \textbf{tell}(d_x + s_x \leq d_y + s_y) \\
\| & (\textbf{abs}\ x, y; \texttt{bf}(x, y))\ (\textbf{abs}\ d_x, s_x; \texttt{box}(x, d_x, s_x)) \\
& \quad\quad (\textbf{abs}\ d_y, s_y; \texttt{box}(y, d_y, s_y))\ \textbf{tell}(s_x + d_x \leq s_y) \\
Persistence \stackrel{\text{def}}{=} & (\textbf{abs}\ x, y; \texttt{in}(x, y))\ \textbf{when}\ \texttt{play}(x)\ \textbf{do next tell}(\texttt{in}(x, y)) \\
& \quad \| \ \textbf{unless}\ \texttt{out}(x, y) \vee \texttt{destroy}(x)\ \textbf{next tell}(\texttt{in}(x, y)) \\
\| & (\textbf{abs}\ x, y; \texttt{bf}(x, y))\ \textbf{when}\ \texttt{play}(y)\ \textbf{do next tell}(\texttt{bf}(x, y)) \\
& \quad \| \ \textbf{unless}\ (\texttt{out}(x, y) \vee \texttt{destroy}(y)\ \textbf{next tell}(\texttt{bf}(x, y)) \\
\| & (\textbf{abs}\ x; \texttt{box}(x, d_x, s_x))\ \textbf{when}\ \texttt{play}(x)\ \textbf{do next tell}(\texttt{box}(x, d_x, s_x)) \\
& \quad\quad \| \ \textbf{unless}\ \texttt{destroy}(x)\ \textbf{next tell}(\texttt{box}(x, d_x, s_x)) \\
Clock(t, v) \stackrel{\text{def}}{=} & \textbf{tell}(t = v) \| \textbf{next}\ Clock(t, v + 1) \\
Play(x, t) \stackrel{\text{def}}{=} & \textbf{when}\ t \geq 1\ \textbf{do tell}(\texttt{play}(x)) \| \textbf{unless}\ t \leq 1\ \textbf{next}\ Play(x, t - 1) \\
Init(t) \stackrel{\text{def}}{=} & (\textbf{wait}\ x; init(x))\ \textbf{do} \\
& \quad (\textbf{abs}\ d_x, s_x; \texttt{box}(x, d_x, s_x)) \\
& \quad\quad Clock(t, 0) \| \textbf{tell}(s_x = t) \| \\
& \quad\quad !\,(\textbf{wait}\ y, d_y, s_y; \texttt{box}(y, d_y, s_y) \wedge s_y \leq t)\ \textbf{do}\ Play(y, d_y) \\
\\
System \stackrel{\text{def}}{=} & (\textbf{local}\ t)\ Init(t) \|! Persistence \|! Constraints \|! BoxOperations \| UsrBoxes
\end{array}
$$

Fig. 2. A utcc model for Dynamic Interactive Scores

The process $Play(x, t)$ adds the constraint $\texttt{play}(x)$ during t time-units. This informs the environment that the box x is currently playing.

The process $Init(t)$ waits until the environment provides the constraint $init(x)$ for the outermost box x to start the execution of the system. Then, the *clock* is started and the start time of x is set to 0. The rest of the boxes wait until their start time is less or equal to the current time (t) to start playing.

Finally, the whole system is the parallel composition between the previously defined processes and the specific user model, e.g. :

$$
\begin{array}{rl}
UsrBoxes \stackrel{\text{def}}{=} & \textbf{tell}(\texttt{mkbox}(a, 22) \wedge \texttt{mkbox}(b, 12) \wedge \texttt{mkbox}(c, 4)) \| \\
& \textbf{tell}(\texttt{mkbox}(d, 5) \wedge \texttt{mkbox}(e, 2)) \| \\
& \textbf{tell}(\texttt{into}(b, a) \wedge \texttt{into}(c, b) \wedge \texttt{into}(d, b) \wedge \texttt{into}(e, d)) \| \\
& \textbf{tell}(\texttt{before}(c, d)) \| \\
& \textbf{whenever}\ \texttt{play}(b)\ \textbf{do unless}\ \texttt{signal}\ \textbf{next} \\
& \quad\quad \textbf{tell}(\texttt{out}(d, b) \wedge \texttt{mkbox}(f, 2) \wedge \texttt{into}(f, a)) \| \\
& \quad\quad \textbf{tell}(\texttt{before}(b, f) \wedge \texttt{before}(f, d))
\end{array}
$$

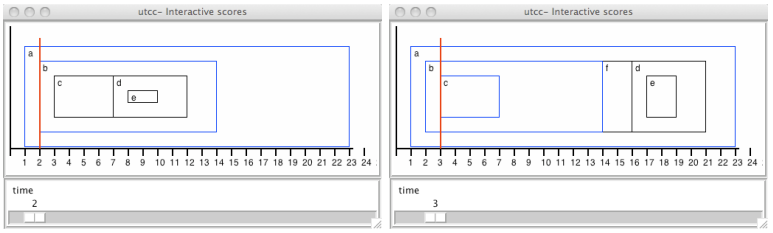
This system defines the hierarchy in Figure 3(a). When b starts playing, the system asks if the signal signal is present (i.e., if it was provided by the environment). If it was not, the box d is taking out from the context b. Furthermore, a new box f is created such that b must be played before f and f before d as in Figure 3(b). Notice that when the box d is taken out from b, the internal box e is still into d preserving its structure.

Verification of the Model The processes defined by the user may lead to situations where the final store is inconsistent as in $st < 5 \wedge st > 7$ where st is the start time of a given box. Take for example the process $UsrBoxes$ above. If the box f is defined with a duration greater than 5, the execution of f (and then that of d) will exceed the boundaries of the box a which contains both structures.

In this context, the declarative view of `utcc` processes as FLTL formulae provides a valuable tool for the verification of the model: The formula $A = [\![P]\!]$ allows us to verify whether the execution of P leads to an inconsistent store. Thus, we can detect pitfalls in the user model such as trying to place a bigger box into a smaller one or taking a box out of the outermost box.

In the following, we present some examples of temporal properties we could verify in an interactive score represented as the process P.

- $[\![P]\!] \vdash \Diamond \exists_{x,d_x,s_x,y,d_y,s_y}(\texttt{box}(x,d_x,s_x) \wedge \texttt{box}(y,d_y,s_y) \wedge \texttt{in}(x,y) \wedge s_x + d_x > s_y + d_y)$: The end time of the box y is less than the end time of the inner box x. I.e., the box y cannot contain x.
- $[\![P]\!] \vdash \forall_x(\exists_{d_x,s_x}(\texttt{box}(x,d_x,s_x) \Rightarrow \Diamond \texttt{play}(x))$: All the musical structures are eventually played.
- $[\![P]\!] \vdash \Box \forall_{x,y}(\texttt{in}(x,y) \wedge \texttt{play}(x) \Rightarrow \texttt{play}(y))$: The execution of the internal box implies the execution of the outer box.
- $[\![P]\!] \vdash \Box \forall_x(\exists_{d_x,s_x} \texttt{box}(x,d_x,s_x) \Rightarrow init(x) \vee \exists_y(\texttt{in}(x,y)))$: Every box is either the initial box or it is contained in another box.
- $[\![P]\!] \vdash \Diamond \forall_x(\exists_{d_x,s_x}(\texttt{box}(x,d_x,s_x) \Rightarrow \texttt{play}(x))$: At some point all the boxes are playing simultaneously.
- $[\![P]\!] \vdash \texttt{signal} \vee \Diamond \texttt{play}(x)$: The signal `signal` is present or else the box x must be played.



(a) (b)

Fig. 3. Example of an Interactive Score Execution

Remark. For the sake of presentation we only defined here the `before` relation. Our model can be straightforwardly extended to support all Allen temporal relations [2]. Making use of the `into` and `out` operations, we can define also the operation $\texttt{move}(a,b)$ meaning, move the structure a into the structure b.

4 A Model for Music Improvisation

As described above, in interactive scores the actual musical output may change depending on interactions with a performer, but the framework is not meant for learning from those interactions, nor to change the score (i.e. improvise) accordingly.

Music improvisation provides a complex context of concurrent systems posing great challenges to modeling tools. In music improvisation, partners behave independently but are constantly interacting with others in controlled ways. The interactions allow building a complex global musical process collaboratively. Interactions become effective when each partner has somehow learned about the possible evolutions of each musical process launched by the others, i.e, their musical *style*. Getting the computer involved in the improvisation process requires learning the musical style of the human interpreter and then playing jointly in the same style. A *style* in this case means some set of meaningful sequences of musical material the interpreter has played. A graph structure called *factor oracle* (FO) is used to efficiently represent this set [1].

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

The oracle (see Figure 4) is learned on-line. For each new input symbol σ_i, a new state i is added and an arrow from $i-1$ to i is created with label σ_i. Starting from $i-1$, the suffix links are iteratively followed backward, until a state is reached where a factor link with label σ_i originates (going to some state j), or until there is no more suffix links to follow. For each state met during this iteration, a new factor link labeled by σ_i is added from this state to i. Finally, a suffix link is added from i to the state j or to state 0 depending on which condition terminated the iteration. Navigating the oracle in order to generate variants is straightforward : starting in any place, following factor links generates a sequence of labelling symbols that are repetitions of portions of the learned sequence; following one suffix link followed by a factor links creates a recombined pattern sharing a common suffix with an existing pattern in the original sequence. This common suffix is, in effect, the musical context at any given time.

In [5] a `tcc` model of FO is proposed. This model has three drawbacks. Firstly, it (informally) assumes the basic calculus has been extended with general recursion in order to correctly model suffix links traversal. Secondly, it assumes dynamic construction of new variables $\delta_{i\sigma}$ set to the state reached by following factor link labelled σ from state i. This construction cannot be expressed with the local variable primitive in basic `tcc`. Thirdly, the model assumes a constraint system over both finite domains and finite sets. We use below the expressive power of the abstraction construction in

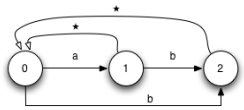


Fig. 4. A FO automaton for $s = ab$

utcc to correct all these drawbacks (see Figure 5). Furthermore, our model leads to a compact representation of the data structure of the FO based on constraints of the form $\texttt{edge}(x, y, N)$ representing an arc between node x and y labeled with N.

$$
\begin{array}{ll}
FO & \stackrel{\text{def}}{=} Counter \parallel Persistence \\
 & \quad \parallel !\,(\textbf{abs}\ Note; \texttt{play}(Note))\ \textbf{whenever}\ ready\ \textbf{do}\ Step_1(Note) \\
Counter & \stackrel{\text{def}}{=} \textbf{tell}(i = 1) \parallel !\,(\textbf{abs}\ x; i = x)\,(\textbf{when}\ ready\ \textbf{do next tell}(i = x + 1) \\
 & \qquad\qquad \parallel \textbf{unless}\ ready\ \textbf{next tell}(i = x)) \\
Persistence & \stackrel{\text{def}}{=} !\,(\textbf{abs}\ x, y, z; \texttt{edge}(x, y, z))\ \textbf{next tell}(\texttt{edge}(x, y, z)) \\
Step_1(Note) & \stackrel{\text{def}}{=} \textbf{tell}(\texttt{edge}(i - 1, i, Note)) \parallel Step_2(Note, i - 1) \\
Step_2(Note, E) & \stackrel{\text{def}}{=} \textbf{when}\ E = 0\ \textbf{do} \\
 & \quad (\textbf{abs}\ k; \texttt{edge}(E, k, Note))\,(\textbf{tell}(\texttt{edge}(i, k, \star)) \parallel \textbf{next tell}(ready)) \\
 & \quad \parallel \textbf{unless}\ \exists_k\, \texttt{edge}(E, K, Note)\ \textbf{next}\ (\textbf{tell}(ready) \parallel \textbf{tell}(\texttt{edge}(i, 0, \star))) \\
 & \textbf{when}\ E \neq 0\ \textbf{do} \\
 & \quad (\textbf{abs}\ j; \texttt{edge}(E, j, \star)) \\
 & \quad\quad \textbf{when}\ \exists_k\, \texttt{edge}(j, k, Note)\ \textbf{do} \\
 & \quad\quad\quad (\textbf{abs}\ k; \texttt{edge}(j, k, Note))\,(\textbf{tell}(\texttt{edge}(i, k, \star)) \parallel \textbf{next tell}(ready)) \\
 & \quad\quad \parallel \textbf{unless}\ \exists_k\, \texttt{edge}(j, k, Note)\ \textbf{next when}\ j \neq 0\ \textbf{do tell}(\texttt{edge}(j, i, Note)) \\
 & \qquad\qquad\qquad\qquad\qquad\qquad \parallel Step_2(Note, j)
\end{array}
$$

Fig. 5. Implementing the FO into utcc

Process *Counter* signals when a new played note can be learned. It can be learned when all links for the previous note have already been added to the FO. Process *Persistence* transmits information about already constructed arcs (factor and suffix) to all future time-units. Process $Step_1$ adds a factor link from $i - 1$ to i labelled with a just played note and launches traversal of suffix links from $i-1$. When state zero is reached by traversing suffix links, process $Step_2$ adds a suffix link from i to a state reached from 0 by a factor link labelled $Note$, if it exists, or from i to state zero, otherwise. For each state k different from zero reached in the suffix links traversal, process $Step_2$ adds factor links labelled $Note$ from k to i.

The inclusion of a new agent in our FO model (e.g. a learner agent for a second performer) entails a new process and new interactions, both with the new process and among the existing ones. In traditional models this usually means major changes in the synchronization scheme, which are difficult to localize and control. In utcc, all synchronization is done semantically, through the available information in the store. Each agent would thus have to be incremented with processes testing for the presence of new information (e.g. a factor link with some label in the other agent's FO graph). The new synchronization behavior that this demands is automatically provided by the blocking ask (abstraction) construct.

5 Concluding Remarks

Here we argued for utcc as a declarative framework for modeling and verifying dynamic multimedia interaction systems. We showed that the synchronization mechanism based on entailment of constraints leads to simpler models that scale up when more agents are added. Moreover, we showed that systems can be formally verified with the underlying temporal logic in utcc. We modeled two non trivial interacting systems. The model proposed for interactive scores in Section 3 improved considerably the expressivity of previous models such as [3]. It allows the composer to dynamically change the structure of the score according to the information derived from the environment.

The results presented here are so far encouraging although much remains to be done at the implementation level. Currently, to guarantee reliable responses in time, we are working on assessing the behavior of utcc processes in real-time contexts. We plan to provide a more principled notion of time where the duration of each time-unit can be related to the amount of computation involved in it. We also plan to enrich our FO model with probabilistic traversals of the graph in the lines of [11].

References

1. C. Allauzen, M. Crochemore, and M. Raffinot. Factor oracle: A new structure for pattern matching. In *Proc. of SOFSEM'99*, LNCS. Springer, 1999.
2. J. F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11), 1983.
3. A. Allombert, G. Assayag, and M. Desainte-Catherine. A system of interactive scores based on Petri nets. In *proceedings of SMC ' 07*, 2007.
4. A. Allombert, G. Assayag, M. Desainte-Catherine, and C. Rueda. Concurrent constraints models for interactive scores. In *proceedings of SMC ' 06*, 2006.
5. G. Assayag, S. Dubnov, and C. Rueda. A concurrent constraints factor oracle model for music improvisation. In *CLEI 06*, 2006.
6. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1991.
7. R. Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, 1999.
8. C. Olarte and C. Rueda. A declarative language for dynamic multimedia interaction systems. Available at http://www.lix.polytechnique.fr/~colarte/, April 2009.
9. C. Olarte and F. D. Valencia. The expressivity of universal timed CCP: Undecidability of monadic FLTL and closure operators for security. In *Proc. of PPDP 08*. ACM, 2008.
10. C. Olarte and F. D. Valencia. Universal concurrent constraint programing: Symbolic semantics and applications to security. In *Proc. of SAC 2008*. ACM, 2008.
11. J. A. Perez and C. Rueda. Non-determinism and probabilities in timed concurrent constraint programming. In *ICLP 2008*. LNCS, 2008.
12. V. Saraswat, R. Jagadeesan, and V. Gupta. Foundations of timed concurrent constraint programming. In *Proc. of LICS'94*. IEEE CS, 1994.
13. V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.