Universal Timed CCP: Expressivity and Applications to Musical Improvisation

Carlos Olarte[1,2], Camilo Rueda[2,3], and Frank D. Valencia[1]

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

Abstract. Universal timed concurrent constraint programming (utcc) is an extension of temporal CCP (tcc) aimed at modeling mobile reactive systems. The language counts with several reasoning techniques such as a symbolic semantics and a compositional semantics based on closure operators. Additionally, utcc processes can be regarded as formulae in first-order linear temporal logic (FLTL). In this paper we first show how the *abstraction* operator introduced in utcc can neatly express arbitrary recursion which is not possible in tcc. Second, we present an encoding of the λ-calculus into utcc. Although utcc has been previously proved to be Turing powerful encoding Minsky machines in it, the encoding we present here is *compositional* unlike that of Minsky machines. Compositionality is an important property of an encoding as it may allow structural analysis in utcc of the source terms; i.e., functional programs. Finally, as compelling application, making use of the recursive definitions in utcc, we model a music improvisation system composed of interactive agents learning a musical style and generating on the fly new material in the same style.

1 Introduction

Concurrent Constraint Programming (CCP) [10] has emerged as a simple but powerful paradigm for concurrency tied to logic. CCP extends and subsumes both concurrent logic programming and constraint logic programming. A fundamental issue in CCP is the specification of concurrent systems by means of constraints, e.g. $x + y \geq 10$, representing partial information about certain variables. Agents in CCP interact with each other by telling and asking information represented as constraints in a global store. More precisely, the process $\mathbf{tell}(c)$ adds the constraint c to the store and the *ask* process $\mathbf{when}\ c\ \mathbf{do}\ P$ executes P if c is entailed by the store. The information in the store growths monotonically in the sense that constraints cannot be dropped.

The timed concurrent constraint programing model (tcc) is a *declarative* framework, closely related to First-Order Linear Temporal Logic (FLTL), for modeling *reactive systems*. In tcc time is conceptually divided into *time intervals* (or *time units*). In a particular time interval, a CCP 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. The resting point determines also a residual process Q which is then executed in the next time interval.

The authors in [8] introduce universal timed CCP (`utcc`). This calculus aims at modeling *mobile* reactive systems. Here mobility is understood as in the π-calculus, i.e., generation and communication of private names. The basic move in `utcc` is to replace the ask process $\mathbf{when}\ c\ \mathbf{do}\ P$ by a *temporary* parametric ask operator of the form $(\mathbf{abs}\ \vec{x}; c)\, P$. Intuitively, this operator executes the process $P[\vec{t}/\vec{x}]$ for every possible term $\vec{t}$ s.t. the current store is able to entail $c[\vec{t}/\vec{x}]$.

In this paper we continue the expressiveness study of `utcc` started in [7]. First we show how the abstraction operator of `utcc` allows to neatly define arbitrary recursion. In particular, we show how recursive functions can be computed in a single time-unit whereas it may take several ones in `tcc`. Next we show that the ability to model mobile behavior in the calculus allows to encode the *call-by-name* λ-calculus. This encoding is built on the encoding proposed in [9] for the π-calculus. Although `utcc` has been previously proved to be Turing powerful encoding Minsky machines [7], the encoding we present here is *compositional* unlike that of Minsky machines.

Finally, as compelling application, making use of the recursive definitions in `utcc`, we model a music improvisation system composed of interactive agents learning a musical style and generating on the fly new material in the same style. An efficient graph structure for learning strings, called *factor oracle* (FO) is implicitly constructed via constraints. We take advantage of the expressivity of the *abstraction* construct of `utcc` to cleanly define recursive traversal of the FO graph and to model its incremental extension using a simple constraint system. The whole FO learning scheme is modeled with standard `utcc` primitives in a simple way, in contrast to the `ntcc` [6] implementation of FO proposed in [3] where features outside the calculus and a dual finite domains and finite sets constraint system had to be used.

2 Preliminaries

CCP-based languages are parametric in a constraint systems which defines the kind of constraints that can be used in the program. A constraint system is defined as follows:

Definition 1 (Constraint System (cs)). *A cs can be represented as a pair* (Σ, Δ) *where* Σ *is a signature and* Δ *is a first-order theory over* Σ. *Given a cs* (Σ, Δ), *let* $\mathcal{L}$ *be its underlying first-order language with variables* $x, y, \ldots$, *and the standard logic symbols* $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists, \forall, \mathtt{true}$ *and* $\mathtt{false}$. Constraints $c, d, \ldots$ *are formulae over* $\mathcal{L}$.

We say that c *entails* d in Δ, written $c \vdash_\Delta d$, iff $(c \Rightarrow d) \in \Delta$ (i.e., iff $c \Rightarrow d$ is true in all models of Δ). We shall omit "Δ" in $\vdash_\Delta$ when no confusion arises. If $d \vdash c$ and $c \vdash d$ we shall write $d \equiv c$. For operational reasons $\vdash$ is often required to be decidable.

We use $\vec{t}$ for a sequence of terms $t_1, \ldots, t_n$ with length $|\vec{t}| = n$. If $|\vec{t}| = 0$ then $\vec{t}$ is written as ϵ. We use $c[\vec{t}/\vec{x}]$, where $|\vec{t}| = |\vec{x}|$ and x_i's are pairwise distinct, to denote c with the free occurrences of x_i replaced with t_i.

2.1 Universal Timed CCP

In [8] the `tcc` calculus is extended for mobile and reactive systems leading to *universal timed* CCP (`utcc`). In `utcc`, the ask operation $\mathbf{when}\ c\ \mathbf{do}\ P$ is replaced with a *parametric ask* 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. From a programming point of view, the variables in $\vec{x}$ can be viewed as the formal parameters of the process P. We shall discuss more about this informal meaning in Section 3.

From a logic perspective, abstractions have a pleasant duality with the local operator: The processes $(\mathbf{local}\,\vec{x}; c)\ P$ and $(\mathbf{abs}\ \vec{x}; c)\ P$ correspond, resp., to the *existential* and *universal* formulae $\exists\vec{x}(c \wedge F_P)$ and $\forall\vec{x}(c \Rightarrow F_P)$ where F_P corresponds to P (see [8] for further details).

Definition 2. *Processes $P, Q, \ldots$ in* `utcc` *are built from constraints in the underlying constraint system by the following syntax:*

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

with the variables in $\vec{x}$ being pairwise distinct.

The process $\mathbf{skip}$ does nothing and $\mathbf{tell}(c)$ adds c to the store in the current time interval. Intuitively, $Q = (\mathbf{abs}\ \vec{x}; c)\ P$ executes $P[\vec{t}/\vec{x}]$ in the current time interval for *all the terms* $\vec{t}$ s.t $c[\vec{t}/\vec{x}]$ is entailed by the store. Q binds the variables $\vec{x}$ in P and c. We denote the set of bound (free) variables in Q by $bv(Q)$ ($fv(Q)$). Furthermore Q evolves into $\mathbf{skip}$ at the end of the time unit, i.e. abstractions are not persistent when passing from one time-unit to the next one. When the set $\vec{x}$ of *abstracted* variables is empty, i.e. ϵ, we recover the `tcc` *ask* process. In what follows we use $\mathbf{when}\ c\ \mathbf{do}\ P$ as a short hand for $(\mathbf{abs}\ \epsilon; c)\ P$. The process $P \parallel Q$ denotes P and Q running in parallel during the current time interval and $(\mathbf{local}\,\vec{x}; c)\ P$ *binds* $\vec{x}$ in P by declaring it private to P under a constraint c. We write $(\mathbf{local}\,\vec{x})\ P$ as short for $(\mathbf{local}\,\vec{x}; \mathtt{true})\ P$. The unit-delay $\mathbf{next}\, P$ executes P in the next time interval. The process $\mathbf{unless}\ c\ \mathbf{next}\, P$ 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. Finally, the *replication* $!\,P$ means $P \parallel \mathbf{next}\, P \parallel \mathbf{next}^{\,2}P...$

Notation 1 *We shall use the derived operator $(\mathbf{wait}\ \vec{x}; c)\ \mathbf{do}\ P$ that* waits*, possibly for several time units until for some $\vec{t}$, $c[\vec{t}/\vec{x}]$ holds. We shall use $\mathbf{whenever}\ c\ \mathbf{do}\ P$ as a shorthand for $(\mathbf{wait}\ \epsilon; c)\ \mathbf{do}\ P$. See [8] for the encoding of* $\mathbf{wait}$.

2.2 Operational Semantics of UTCC

The structural operational semantics (SOS) of `utcc` considers *transitions* between process-store *configurations* $\langle P, c\rangle$ with stores represented as constraints and processes quotiented by $\equiv$. We use $\gamma, \gamma', \ldots$ to range over configurations.

Definition 3. *Let $\equiv$ be the smallest congruence satisfying: (0) $P \equiv Q$ if they differ only by a renaming of bound variables (1) $P \parallel \mathbf{skip} \equiv P$, (2) $P \parallel Q \equiv Q \parallel P$, (3) $P \parallel (Q \parallel R) \equiv (P \parallel Q) \parallel R$, (4) $P \parallel (\mathbf{local}\,\vec{x}; c)\, Q \equiv (\mathbf{local}\,\vec{x}; c)\,(P \parallel Q)$ if $\vec{x} \notin fv(P)$, (5) $(\mathbf{local}\,\vec{x}; c)\,\mathbf{skip} \equiv \mathbf{skip}$. Extend $\equiv$ by decreeing that $\langle P, c\rangle \equiv \langle Q, c\rangle$ iff $P \equiv Q$.*

The SOS transitions are given by the relations $\longrightarrow$ and $\Longrightarrow$ in Table 1. The *internal* transition $\langle P, d\rangle \longrightarrow \langle P', d'\rangle$ should be read as "P with store d reduces, in one internal step, to P' with store d' ". The *observable transition* $P \xRightarrow{(c,d)} R$ should be read as "P on input c, reduces in one *time unit* to R and outputs d". The observable transitions are obtained from finite sequences of internal transitions. The transitive and reflexive closure of $\longrightarrow$ and $\Longrightarrow$ are written as $\longrightarrow^*$ and $\Longrightarrow^*$, respectively.

$$\mathrm{R_T}\ \frac{}{\langle \mathbf{tell}(c), d\rangle \longrightarrow \langle \mathbf{skip}, d \wedge c\rangle} \qquad \mathrm{R_P}\ \frac{\langle P, c\rangle \longrightarrow \langle P', d\rangle}{\langle P \parallel Q, c\rangle \longrightarrow \langle P' \parallel Q, d\rangle}$$

$$\mathrm{R_L}\ \frac{\langle P, c \wedge (\exists \vec{x} d)\rangle \longrightarrow \langle P', c' \wedge (\exists \vec{x} d)\rangle}{\langle (\mathbf{local}\,\vec{x}; c)\, P, d\rangle \longrightarrow \langle (\mathbf{local}\,\vec{x}; c')\, P', d \wedge \exists \vec{x} c'\rangle}$$

$$\mathrm{R_U}\ \frac{d \vdash c}{\langle \mathbf{unless}\, c\, \mathbf{next}\, P, d\rangle \longrightarrow \langle \mathbf{skip}, d\rangle} \qquad \mathrm{R_R}\ \frac{}{\langle !\, P, d\rangle \longrightarrow \langle P \| \mathbf{next}\, !\, P, d\rangle}$$

$$\mathrm{R_A}\ \frac{d \vdash c[\vec{t}/\vec{x}] \quad |\vec{t}| = |\vec{x}|}{\langle (\mathbf{abs}\,\vec{x}; c)\, P, d\rangle \longrightarrow \langle P[\vec{t}/\vec{x}] \| (\mathbf{abs}\,\vec{x}; c \wedge \vec{x} \neq \vec{t})\, P, d\rangle}$$

$$\mathrm{R_S}\ \frac{\gamma_1 \longrightarrow \gamma_2}{\gamma_1' \longrightarrow \gamma_2'}\ \text{if } \gamma_1 \equiv \gamma_1' \text{ and } \gamma_2 \equiv \gamma_2'$$

$$\mathrm{R_O}\ \frac{\langle P, c\rangle \longrightarrow^* \langle Q, d\rangle \not\longrightarrow}{P \xRightarrow{(c,d)} F(Q)}$$

Table 1. Internal and observable reductions. $\equiv$ and F are given in Definitions 3 and 4. In $\mathrm{R_A}$, $\vec{x} \neq \vec{t}$ denotes $\bigvee_{1 \leq i \leq |\vec{x}|} x_i \neq t_i$. If $|\vec{x}| = 0$, $\vec{x} \neq \vec{t}$ is defined as `false`.

We only describe some of the rules in Table 1 due to space restrictions. The other rules are standard, easily seen to realize the operational intuitions given above (see [6] for further details). As clarified below, the seemingly missing rules for "next" and "unless" processes are given by $\mathrm{R_O}$.

Rule $\mathrm{R_A}$ describes the behavior of $(\mathbf{abs}\ \vec{x}; c)\, P$. If the store entails $c[\vec{t}/\vec{x}]$ then $P[\vec{t}/\vec{x}]$ is executed. Additionally, the abstraction persists in the current time interval to allow other potential replacements of $\vec{x}$ in P but c is augmented with $\vec{x} \neq \vec{t}$ to avoid executing $P[\vec{t}/\vec{x}]$ again.

Rule $\mathrm{R_O}$ says that an observable transition from P labeled with (c, d) is obtained from a terminating sequence of internal transitions from $\langle P,\ c\rangle$ to a $\langle Q, d\rangle$. The process to be executed in the next time interval is equivalent to $F(Q)$ (the "future" of Q). $F(Q)$ is obtained by removing from Q abstractions and any local information which has been stored in Q, and by "unfolding" the sub-terms within "next" and "unless" expressions.

Definition 4. *Let F be a partial function defined as:*

$F(\mathbf{skip}) = \mathbf{skip}$	$F((\mathbf{abs}\,\vec{x}; c)\, Q) = \mathbf{skip}$
$F(P_1 \parallel P_2) = F(P_1) \parallel F(P_2)$	$F((\mathbf{local}\,\vec{x}; c)\, Q) = (\mathbf{local}\,\vec{x}; \mathtt{true})\, F(Q)$
$F(\mathbf{next}\, Q) = Q$	$F(\mathbf{unless}\, c\, \mathbf{next}\, Q) = Q$

Definition 5 (Input-output Behavior). *Let* $s = c_1...c_i$ *and* $s' = c'_1...c'_i$ *be sequences of constraints. If* $P = P_1 \xrightarrow{(c_1,c'_1)} P_2 \xrightarrow{(c_2,c'_2)} ...P_i \xrightarrow{c_i,c'_i}$, *we write* $P \xrightarrow{(s,s')}$. *The set* $io(P) = \{(s, s') \mid P \xrightarrow{(s,s')}\}$ *denotes the* input-output *behavior of* P.

3 Recursion in UTCC

In this section we show that the abstraction operator in `utcc` is able to model arbitrary recursion. It allows us to compute recursive functions in a single time-unit which is not possible in `tcc`. Given the pairwise distinct variables $\vec{x}$, let $P(\vec{x}) \stackrel{\text{def}}{=} A$ be a (recursive) procedure definition where A is a process possibly containing *calls* of P of the form $P(\vec{y})$ with $|\vec{x}| = |\vec{y}|$. As expected, the call of P with *actual* parameters $\vec{y}$ syntactically replaces the *formal parameters* $\vec{x}$ in A, thus executing $A[\vec{y}/\vec{x}]$. Furthermore, to avoid dynamic scoping behavior we restrict $fv(A) \subseteq \{x_1, ..., x_n\}$ (see [6]).

3.1 Recursion in TCC-calculi

In `tcc`-based languages only a basic form of recursion is allowed, namely value-passing recursion. Here in a recursive call of the form $P(x)$, it is assumed that the current store d entails $x = t$ for some rigid (constant) term t. To keep bounded the response time of the system, recursive calls in the body of a procedure definition must be guarded by a **next** process. Recursive definition can be encoded as follows:

$$\ulcorner P(x) \stackrel{\text{def}}{=} A \urcorner = !\,\mathbf{when}\ \mathtt{call}(p)\ \mathbf{do}\ (\mathbf{local}\, x)\,(x \leftarrow parg \parallel \widehat{A})$$

where: (1) p and $pargs$ don't occur free in A. (2) $call(p)$ is the constraint $p = 1$. (3) $x \leftarrow t$ makes persistent the assignment $x = t$ in all time-units (i.e. $!\,\mathbf{tell}(x = t)$). (4) $\widehat{A}$ replaces in A every recursive call $P(t)$ by $\mathtt{call}(p) \parallel pargs = t$.

A call $p(t)$ from a process is encoded as: $(\mathbf{local}\, p, pargs)\ (\ulcorner P(x) \stackrel{\text{def}}{=} A \urcorner \parallel \widehat{p(t)})$

Alternative Recursive Definitions in `tcc` In [5] the authors explore different extensions of `tcc` to handle recursive definitions. The languages considered in the study are: `rep` (considers only replication, i.e. there is no recursive definitions) ; $\mathtt{rec}_p$ (in all recursive definition $P(\vec{x}) \stackrel{\text{def}}{=} A$, $fv(A) \subseteq \vec{x}$) ; $\mathtt{rec}_i$ (as $\mathtt{rec}_p$ but actual parameters in recursive calls are identical to the formal parameter) ; $\mathtt{rec}_d$ (procedures without parameters and free variables in procedure bodies with dynamic scoping). And $\mathtt{rec}_s$(similar to $\mathtt{rec}_d$ but considering static scoping).

The main result in [5] is that $\mathtt{rec}_p$ and $\mathtt{rec}_d$ are equally expressive and strictly more expressive than the other languages. In this paper we are considering a variant of `utcc` including recursive definition adhering to the condition of $\mathtt{rec}_p$. Since the encodings from $\mathtt{rec}_i$, $\mathtt{rec}_s$ into $\mathtt{rec}_p$ [5] uses the deterministic fragment of `ntcc` which is a sub-calculus of `utcc`, it is trivial that these other forms of recursion can be encoded in `utcc`.

3.2 Recursive Definition in `utcc`

The abstraction operator in `utcc` gives an interesting way to encode recursion in the language. Assume the recursive definition $P(\vec{x}) \stackrel{\texttt{def}}{=} A$. Let p be an uninterpreted predicate of arity $|\vec{x}|$. The encoding of the procedure definitions and calls is as follows:

- $\ulcorner P(x) \stackrel{\texttt{def}}{=} A \urcorner = (\mathbf{abs}\ x; p(\vec{x}))\, \widehat{A}$ where $\widehat{A}$ is the process A where each recursive call of the form $P(\vec{t})$ is replaced by $\mathbf{tell}(p(\vec{t}))$.
- A call to P of the form $P(\vec{t})$ is replaced by $\mathbf{tell}(p(\vec{t}))$

Notice how this encoding reflects our informal meaning of abstractions as having formal parameters of a process. The guard ("c") of the abstraction plays here the role of a pre-condition the actual parameters must satisfy in order to execute P.

A simple example Due to the restriction of next-guarding the recursive calls in a procedure definition, `tcc`-based languages cannot compute a recursive functions in a single time-unit. Here we show how the encoding above using the abstraction operator in `utcc` overcomes this problem. Assume the following procedure computing $n!$:

$$\begin{array}{ll} Fact(N,X) & := Fact'(N,1,X) \\ Fact'(N,M,X) & := \text{if } N <= 1 \text{ then } X = M \\ & \quad\ \text{else } Fact'(N-1, M*N, X) \end{array}$$

where X is a parameter by reference in which the result will be stored. These procedures can be encoded as follows:

$$\begin{array}{ll} !\,(\mathbf{abs}\ N,X; fact(N,X)) & \mathbf{tell}(fact'(N,1,X)) \\ !\,(\mathbf{abs}\ N,M,X; fact'(N,M,X)) & \mathbf{when}\ N \leq 1\ \mathbf{do}\ \mathbf{tell}(X=M)\ \| \\ & \mathbf{when}\ N > 1\ \mathbf{do}\ \mathbf{tell}(fact'(N-1, N \times M, X)) \end{array}$$

Executing the process above in parallel with $\mathbf{tell}(fact(3,X))$, we observe the constraints $\{fact(3,X), fact'(3,1,X), fact'(2,3,X), fact'(1,6,X), X{=}6\}$ leading to the expected result.

3.3 Expressiveness results

In [11] is shown that only Lustre [4] programs with finite domain variables can be encoded into `tcc`. The reason is the "weak" mechanisms `tcc` offers to transmit the information among time units. Let $D = \{d_1, ..., d_i\}$ be the domain of x. The following process passes the value of x to the next time unit:

$$\mathbf{when}\ x{=}d_1\ \mathbf{do}\ \mathbf{next}\ \mathbf{tell}(x{=}d_1) \| ... \| \mathbf{when}\ x{=}d_i\ \mathbf{do}\ \mathbf{next}\ \mathbf{tell}(x{=}d_i)$$

Notice that D must be finite. In `utcc` we can do the same as follows:

$$(\mathbf{abs}\ d; x = d)\, \mathbf{next}\ \mathbf{tell}(x = d)$$

Therefore, based on the expressiveness study in [11] we can conclude that `utcc` can encode Lustre programs regardless the domain of the variables.

4 Encoding the λ-calculus into UTCC

In this section we give a *compositional* encoding of the λ-calculus into `utcc` processes. The encoding shows how `utcc` is able to mimic one of the most notable and simple computational models achieving Turing completeness. Here, the ability to express mobility is central to our encoding which is built upon the ideas in the encodings in [9] for the π-calculus. Our interest in this encodings is twofold: For `utcc` is a significant test of expressiveness and it can be the basis for the study of languages combining concurrent constraint programming and functional programming (e.g., [2]).

Terms in the λ-calculus denoted by $M, N, ...$ are built from variables $x, y, ...$ by the following syntax:

$$M ::= x \mid (\lambda x.M) \mid (MN)$$

The term $(\lambda x.M)$ is known as λ-abstraction and (MN) as the *application* of M to N.

Computation in the *call-by-name* λ-calculus is described by the relation $\longrightarrow_\lambda$:

$$\beta \ \frac{}{((\lambda x.M)N) \longrightarrow_\lambda M[N/x]} \quad \mu \ \frac{M \longrightarrow_\lambda M'}{MN \longrightarrow_\lambda M'N}$$

Rule β replaces the placeholder x by the argument N in the body M. The parameter N in such an expression is *not evaluated* before the substitution takes place. The expression $(\lambda x.M)$ is called β-redex and the result of the reduction, $M[N/x]$, is called *contractum*. Rule μ allows to replace the leftmost β-redex in the *application* (MN) by its contractum. Notice that terms in the body of an abstraction are not reduced.

In our encoding we shall mimic the input and output actions in the π-calculus encoding of the λ-calculus. Notice that inputs and outputs in the π-calculus disappear only after being involved in an input-output interaction. In `utcc`, the tell and abstraction processes can be thought of as being outputs and inputs, resp., in π, but they are not automatically transferred from one time-unit to the next one—intuitively, they will disappear right after the current time unit even if they did not interact.

Consequently, to mimic π inputs we shall use the process $\mathbf{wait}$ (Notation 1) which transfers itself from one time-unit to the next one until for some t, $c[t/x]$ is entailed by the current store—intuitively, after interacting with an output in the store. To mimic π outputs, we require a derived constructor able to perform the output until some process is able to *"read"* the constraint produced—after interacting with an input process. We shall write $\underline{\mathbf{tell}}(c)$ for the persistent output of c until some process reads c. We also define an auxiliary input process that adds a constraint acknowledging the reading of c namely $(\underline{\mathbf{wait}}\ \vec{x}; c)\ \mathbf{do}\ P$. Assuming $stop, go \notin fv(c)$, these processes are defined as:

$$\begin{aligned}
\underline{\mathbf{tell}}(c) \quad & \stackrel{\text{def}}{=} (\mathbf{local}\, go, stop)\ \mathbf{tell}(\mathtt{out}'(go)) \parallel !\ \mathbf{when}\ \mathtt{out}'(go)\ \mathbf{do}\ \mathbf{tell}(c) \parallel \\
& \quad !\ \mathbf{unless}\ \mathtt{out}'(stop)\ \mathbf{next}\ \mathbf{tell}(\mathtt{out}'(go)) \parallel !\ \mathbf{when}\ \overline{c}\ \mathbf{do}\ !\ \mathbf{tell}(\mathtt{out}'(stop)) \\
(\underline{\mathbf{wait}}\ \vec{x}; c)\ \mathbf{do}\ P & \stackrel{\text{def}}{=} (\mathbf{wait}\ \vec{x}; c)\ \mathbf{do}\ (P \parallel \mathbf{tell}(\overline{c}))
\end{aligned}$$

Constraint $\overline{c}$ represents an acknowledge that constraint c was read.

The encoding below maps arbitrary λ-terms into `utcc` processes. We presuppose a constraint system with two uninterpreted predicates $\mathtt{out}_2$ and $\mathtt{out}_3$ and the corresponding acknowledgments $\overline{\mathtt{out}_2}$ and $\overline{\mathtt{out}_3}$. For the sake of simplicity, we shall omit the sub-indexes in $\texttt{out}_2$ and $\texttt{out}_3$ and they are understood as the arity of $\texttt{out}$.

$$\begin{array}{lll}
\text{variable} & [\![x]\!]_u & \stackrel{\text{def}}{=} \underline{\mathbf{tell}}(\texttt{out}(x,u)) \\
\lambda\text{-abstraction} & [\![\lambda x.M]\!]_u & \stackrel{\text{def}}{=} (\underline{\mathbf{wait}}\ x,v; \texttt{out}(u,x,v))\ \mathbf{do\ next}\ [\![M]\!]_v \\
\text{Application} & [\![(MN)]\!]_u & \stackrel{\text{def}}{=} (\mathbf{local}\ v)\ ([\![M]\!]_v \parallel (\mathbf{local}\ w)\ \underline{\mathbf{tell}}(\texttt{out}(v,w,u) \parallel \\
 & & \qquad !\ (\underline{\mathbf{wait}}\ m; \texttt{out}(w,m))\ \mathbf{do\ next}\ [\![N]\!]_m))
\end{array}$$

Finally, the following theorem states the correctness of our encoding.

Theorem 1 (Correctness). *Let M and N be λ terms.*

- (Soundness)*. If $M \longrightarrow_\lambda N$, there is P s.t. $[\![M]\!]_u \Longrightarrow^* P$ and $P \sim^{io} [\![N]\!]_u$.*
- (Completeness)*. If $[\![M]\!]_u \Longrightarrow^* P$, there is N' s.t $M \longrightarrow_\lambda^* N'$ and $[\![N']\!]_u \sim^{io} P$.*

5 Application in Music Improvisation

Music improvisation provides a complex context of concurrent systems that poses 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 in this setting when each partner has somehow learned about the possible evolutions of each musical process launched by the other musicians (i.e. their musical *style*). Getting the computer involved in the improvisation process requires that it must first learn the musical style of the human interpreter and then begin to play jointly in the same style. A *style* in this case means some set of meaningful sequences of musical material (notes, durations, etc.) the interpreter has played. A graph structure called *factor oracle* (FO) is used to efficiently represent this set.

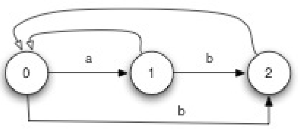

Fig. 1. A FO automaton for $s = ab$

A FO (see [1]) is a finite state automaton constructed in an incremental fashion. A sequence of symbols $s = \sigma_1\sigma_2 \ldots \sigma_n$ is learned in such an automaton, which states are $0, 1, 2 \ldots n$. There is always a transition arrow (called factor link) labeled by 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 1) is learned on-line. For each new entering 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 [3] a `ntcc` 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 also 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 really be expressed with the local variable primitive in basic `ntcc`. 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 `utcc` to correct all these drawbacks (see Figure 2).

$$
\begin{array}{lll}
\mathit{FO} & \stackrel{\text{def}}{=} & \mathit{Counter} \parallel \mathit{Persistence} \\
 & & \parallel ! \, (\mathbf{abs}\ \mathit{Note}; \mathtt{play}(\mathit{Note}))\ \mathbf{whenever}\ \mathit{ready}\ \mathbf{do}\ \mathit{Step}_1(\mathit{Note}) \\
 & & \\
\mathit{Counter} & \stackrel{\text{def}}{=} & \mathbf{tell}(i=1) \parallel ! \, (\mathbf{abs}\ x; i=x)\ (\mathbf{when}\ \mathit{ready}\ \mathbf{do}\ \mathbf{next}\ \mathbf{tell}(i=x+1) \\
 & & \qquad\qquad\qquad\qquad\qquad \parallel \mathbf{unless}\ \mathit{ready}\ \mathbf{next}\ \mathbf{tell}(i=x)) \\
 & & \\
\mathit{Persistence} & \stackrel{\text{def}}{=} & ! \, (\mathbf{abs}\ x,y,z; \mathtt{edge}(x,y,z))\ \mathbf{next}\ \mathbf{tell}(\mathtt{edge}(x,y,z)) \\
 & & \\
\mathit{Step}_1(\mathit{Note}) & \stackrel{\text{def}}{=} & \mathbf{tell}(\mathtt{edge}(i-1,i,\mathit{Note})) \parallel \mathit{Step}_2(\mathit{Note},i-1) \\
 & & \\
\mathit{Step}_2(\mathit{Note},E) & \stackrel{\text{def}}{=} & \mathbf{when}\ E=0\ \mathbf{do} \\
 & & \quad (\mathbf{abs}\ k; \mathtt{edge}(E,k,\mathit{Note}))\ (\mathbf{tell}(\mathtt{edge}(i,k,\star)) \parallel \mathbf{next}\ \mathbf{tell}(\mathit{ready})) \\
 & & \quad \parallel \mathbf{unless}\ \exists_k\ \mathtt{edge}(E,K,\mathit{Note})\ \mathbf{next}\ (\mathbf{tell}(\mathit{ready}) \parallel \mathbf{tell}(\mathtt{edge}(i,0,\star))) \\
 & & \mathbf{when}\ E \neq 0\ \mathbf{do} \\
 & & \quad (\mathbf{abs}\ j; \mathtt{edge}(E,j,\star)) \\
 & & \qquad \mathbf{when}\ \exists_k\ \mathtt{edge}(j,k,\mathit{Note})\ \mathbf{do} \\
 & & \qquad\quad (\mathbf{abs}\ k; \mathtt{edge}(j,k,\mathit{Note}))\ (\mathbf{tell}(\mathtt{edge}(i,k,\star)) \parallel \mathbf{next}\ \mathbf{tell}(\mathit{ready})) \\
 & & \qquad \parallel \mathbf{unless}\ \exists_k\ \mathtt{edge}(j,k,\mathit{Note})\ \mathbf{next}\ \mathbf{when}\ j \neq 0\ \mathbf{do}\ \mathbf{tell}(\mathtt{edge}(j,i,\mathit{Note})) \\
 & & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \parallel \mathit{Step}_2(\mathit{Note},j)
\end{array}
$$

Fig. 2. Implementing the FO into `utcc`

Process *Counter* signals when a new played note can be learned. A new note 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 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.

6 Concluding Remarks

In this paper we show how arbitrary recursion can be encoded in the utcc calculus. Furthermore, we give a utcc process able to transfer the value of a variable from a time-unit to the next one regardless the cardinality of the domain of the variable. It allows us to state that utcc is able to encode arbitrary Lustre programs based on the study in [11]. We go further and we prove that the *call-by-name* λ-calculus can be encoded into utcc. As a compelling applications we implement in utcc a music improvisation system. The solution we propose overcomes the drawbacks of the previous implementation in ntcc [3]. Here the abstraction operator and the ability it provides to transmit information among time-units is central to the simplicity of the solution proposed.

Currently we have implemented in Oz (http://www.mozart-oz.org/) an interpreter of the utcc calculus. In this we were able to test and run the program presented in Section 5. We are planning to improve the implementation of the interpreter to deal with *soft* real time which is required to develop real applications in this setting.

References

1. R. M. Allauzen C., Crochemore M. Factor oracle: a new structure for pattern matching'. In *Proc. of SOFSEM'99*. LNCS, 1999.
2. M. Alpuente, B. Gramlich, and A. Villanueva. A framework for timed concurrent constraint programming with external functions. *Electr. Notes Theor. Comput. Sci.*, 188, 2007.
3. G. A. C. Rueda and S. Dubnov. A concurrent constraints factor oracle model for music improvisation. In *Proc. of CLEI 2006*, 2006.
4. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.
5. M. Nielsen, C. Palamidessi, and F. Valencia. On the expressive power of temporal concurrent constraint programming languages. In *Proc. of PPDP'02*. IEEE Computer Society, 2002.
6. M. Nielsen, C. Palamidessi, and F. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. *Nordic Journal of Computing*, 9(1), 2002.
7. 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*, 2008.
8. C. Olarte and F. D. Valencia. Universal concurrent constraint programing: Symbolic semantics and applications to security. In *Proc. of SAC 2008*. ACM, 2008.
9. J. P. R. Milner and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100, Sept. 1992.
10. V. A. Saraswat. *Concurrent Constraint Programming*. MIT Press, 1993.
11. S. Tini. On the expressiveness of timed concurrent constraint programming. *Electr. Notes Theor. Comput. Sci.*, 27, 1999.