Antescofo Intermediate Representation

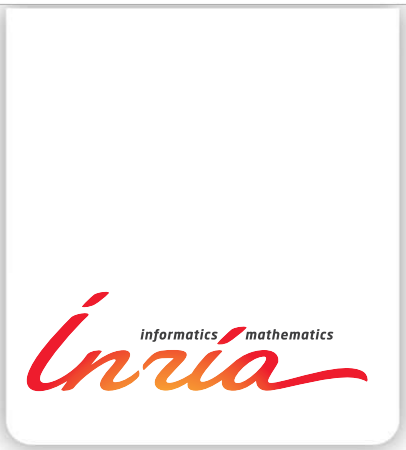
Florent Jacquemard, Clément Poncelet Sanchez

Antescofo Intermediate Representation

Florent Jacquemard, Clément Poncelet

RESEARCH REPORT
N° 8520
April 2014
Project-Teams MuTant

ISSN 0249-6399 ISRN INRIA/RR--8520--FR+ENG



Antescofo
Intermediate Representation

Florent Jacquemard*†, Clément Poncelet‡†

Project-Teams MuTant

Research Report n° 8520 — April 2014 — 12 pages

Abstract: We describe an intermediate language designed as a medium-level internal representation of programs of the interactive music system Antescofo. This representation is independent both of the Antescofo source language and of the architecture of the execution platform. It is used in tasks such as verification of timings, model-based conformance testing, static control-flow analysis or simulation.
This language is essentially a flat representation of Antescofo's code, as a finite state machine extended with local and global variables, with delays and with concurrent threads creation. It features a small number of simple instructions which are either blocking (wait for external event, signal or duration) or not (variable assignment, message emission and control).

Key-words: Interactive Music Systems, Compilation, Formal Models

This work has been supported by the ANR project Inedit (ANR-12-CORD-009) `http://inedit.ircam.fr`.

* INRIA
† Ircam, UMR STMS 9912 CNRS/UPMC
‡ DGA & INRIA

RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Représentation Intermédiaire pour le langage d'Antescofo

Résumé : Ce rapport décrit un langage intermédiaire conçu pour la représentation interne de programmes du système musical interactif Antescofo. Il est actuellement utilisé dans des tâches de vérification portant en particulier sur les durées, de test de conformité fondé sur modèles, d'analyse statique et de simulation.

Mots-clés : Systèmes musicaux interactifs, compilation, modèles formels

1 Intermediate Code: Syntax

We describe in this section an abstract syntax for the intermediate code which will be the result of a front-end compilation of Antescofo's programs. It is defined independently of Antescofo's source language and of the architecture of the execution platform. We give in the description some examples corresponding to the compilation of programs in Antescofo language.

1.1 Values

1.1.1 Atomic Values

We assume the same scalar values as in Antescofo, see [5, 2]: Booleans values *true* and *false*, the integers, the floats (double), the strings and one undefined value (which is not used in this document). We also assume compounds values for vectors and maps.

Durations are a specific type of value. They can be expressed with different time units, corresponding to different clocks. For instance, the seconds is the time unit of the wall clock (physical time). In Antescofo, the most important time unit is beats, which refers to an inferred tempo.

As explained in Section 2.1, we assume that time units are inter-convertible and hence we shall sometimes drop them in the expression of delays in the following.

1.1.2 Variables

Let $\mathcal{X}_g$ and $\mathcal{X}_l$ be two disjoint infinite sets of respectively *global variables* and *local variables*.

1.1.3 Expressions

The expressions are the same as in the Antescofo language [5]. Note that predicates operating on duration values will rely on multi clock services described in Section 2.1. An expression is called *ground* when it does not contain variables.

1.2 Symbols

1.2.1 Input Symbols

We assume a given set of input symbols $\mathcal{I} = \{i_0, \ldots\}$, called *input events*, representing some information expected from the external environment. The set $\mathcal{I}$ is assumed totally ordered by a function called *next*.

We take for instance a set of Antescofo's *events* (notes etc), as defined in Section 2 of [5], together with their positions in the score. For such an input symbol i at position n, $next(i)$ is then defined as the event at position $n + 1$.

A generalization of the total ordering on input events into a DFA with state set $\mathcal{Q}$ and input alphabet $\mathcal{I}$ will be the subject of further work.

1.2.2 Output Symbols

We assume a given set of symbols $\mathcal{O} = \{a_0, a_1 \ldots\}$, representing action emitted or messages sent to the external environment.

For Antescofo, the elements of $\mathcal{O}$ are called *internal* (atomic) *actions* and can be messages to MAX/MSP, OSC messages...

1.2.3 Signals

We consider internal *signals* represented by natural numbers, and denoted s...

In the case of Antescofo, typical signals include the name of groups, kill signals and signals associated to missed events (similar to exceptions).

1.3 Machines

A *machine* $\mathcal{M}$ is an table of fixed size containing instructions in the set presented below. A *location* ℓ is an index in the table (natural number). We assume a fixed total ordering $\ll$ on locations of $\mathcal{M}$. It will be used to reflect the order of instructions in the source Antescofo program. Therefore, the ordering $\ll$ may differ from the ordering on natural numbers. However, for the sake of readability, we write $\ell + 1$ for the the successor of ℓ wrt $\ll$.

1.4 Instructions

We now enumerate the instructions of the intermediate code, with informal descriptions (Section 2 provides a detailed definition of semantics).

Every instruction has an implicit *source location* ℓ which is its index in the table $\mathcal{M}$. It can have zero, one or several *target location* denoted ℓ'...

We consider two categories of instructions. The *synchronous* instructions are instantaneous: they are executed simultaneously, in a single logical instant. The *asynchronous* instructions are blocking: they stop the computation, waiting for an event to happen. Time is flowing while waiting during the execution of an asynchronous instruction κ, and the date of the event unlocking κ defines a new logical instant, as explained in Section 2.5.

1.4.1 Atomic Synchronous Instructions

All these instructions are executed within the same logical instant.

emit s, where s is an internal signal. Signal emission: broadcast the signal s, and continue at $\ell + 1$ with the next instruction.

send a, where $a \in \mathcal{O}$ is an output symbol. Message sending: send a to the external environment (e.g. OSC or MAX message), and continue at $\ell + 1$ with the next instruction.

$x := e$, where x is a local or global variable. Variable assignment.

stop. Terminates the execution.

1.4.2 Branching Synchronous Instruction

if e jump ℓ'. Conditional: if the Boolean expression e evaluates to true, then jump to the location ℓ', otherwise continue at $\ell + 1$ with the next instruction.

1.4.3 Concurrent Synchronous Instructions

The two following instructions start a concurrent execution, with passing or not of the local environment.

spawn ℓ'. Continue the current thread with the next instruction at $\ell + 1$, and start concurrently a new thread at location ℓ' with a copy of the local environment.

ℓ	sustain $(\ell'-3)\,(\ell'-1)$	$\ell'-3$	spawn$_0$ ℓ'	ℓ'	code of the loop
$\ell+1$	next instruction	$\ell'-2$	await e jump $\ell'-3$	⋮	
⋮		$\ell'-1$	await e' jump $\ell'+k$	$\ell'+k$	stop

Figure 1: Encoding of repeat e jump ℓ' for e'

spawn$_0$ ℓ'. Continue the current thread with the next instruction at $\ell+1$, and start concurrently a new thread at location ℓ' with an new empty local environment.

1.4.4 Atomic Asynchronous Instructions

The following instructions let the time flow. Each of them has an explicit target location ℓ'.

await e jump ℓ', where e is an expression that must be evaluable in a duration value d in a time unit tu. Wait for d units of the time units tu, and jump to location ℓ'.

(opt) repeat e jump ℓ' for e', where e and e' are expressions that must be evaluable in duration values d and d' in respective time units tu and tu'. Periodically wait for d units of the time unit tu, and at each iteration, create a new thread at location ℓ'. Stop iterating after d' units of the time unit tu'.

receive i jump ℓ', where $i \in \mathcal{I}$ is an input event. Wait for the reception of the input event i and jump to location ℓ'.

present s jump ℓ', where s is a signal. Wait for s and jump to location ℓ'.

suspend e jump ℓ', where e is a boolean expression. Wait for e to become evaluable to true and then jump to location ℓ'.

Note the difference between the synchronous if and the asynchronous suspend: The former evaluates immediately the associated expression (with failure when it is not evaluable) whereas the latter blocking instruction waits until the expression is evaluable to true.

The instruction repeat can be encoded using a combination of sustain, await and spawn$_0$, see Figure 1.4.4, but it is more efficient to use this instruction which rely on a special clock service described in Section 2.1, and avoids to start a timer at each iteration.

1.4.5 Branching Asynchronous Instruction

asap L, where L is a non-empty list $\ell_1 \dots \ell_n$ of locations of asynchronous transitions in $\mathcal{M}$. Wait concurrently (competitively) for the atomic asynchronous instructions $\mathcal{M}(\ell_1)$, $\dots \mathcal{M}(\ell_n)$. Once one instruction $\mathcal{M}(\ell_i)$ in unlocked, jump to its target. The other instructions are discarded.

sustain $\ell_1\,\ell_2$, where ℓ_2 is the location of an asynchronous instruction in $\mathcal{M}$. Every asynchronous instruction κ following ℓ_1 will be controlled by $\mathcal{M}(\ell_2)$. If κ is unlocked before $\mathcal{M}(\ell_2)$, then the execution continues at the target of κ, but $\mathcal{M}(\ell_2)$ is not discarded (unlike with asap). If $\mathcal{M}(\ell_2)$ is unlocked before κ, jump to its target and κ is discarded.

The instruction sustain could be encoded by adding ℓ_2 (in asap instructions) to every asynchronous instruction following ℓ_1. It has been added to lighten notations, and is similar to the construction of hierarchical states (see e.g. [6], [8], [4]).

2 Intermediate Code: Semantics

We present in this section the execution of a machine $\mathcal{M}$. It follows reactive synchronous semantics, with concurrent thread creation and cooperative multitasking. It extends previous works on the timed-automata based definition of an operational semantics of the static kernel of Antescofo [1].

Intuitively, the machine $\mathcal{M}$ is ran by several concurrent "threads", organized in a tree structure (called *global tree*). Each thread (called *local state* in Section 2.2) points to a line ℓ in $\mathcal{M}$. There is also a global store γ, for assignment of global (shared) variables, not attached to a particular thread. One step of execution of M, at instant t_k consists in the following successive steps.

1. For every thread, iteratively execute the pointed instruction as long as it is *synchronous*. The order of execution is defined after $\ll$ (see Section 1.3). The executions are assumed instantaneous (hypothesis of *synchronicity*): the date is still t_k during the execution of all successive synchronous instructions. When done (*i.e.* after step 1 and before step 2) every thread points to an asynchronous instruction.
2. Wait, during a delay d, for a *logical event*, which can be
   - a signal sent or a global variable modified during step 1 (in this case $d = 0$)
   - an external input event
   - an external modification of a global variable
   - the expiration of a delay (following an instruction `await` or `repeat`).

   Then execute the unlocked (asynchronous) instructions (do `jump`'s) and reorganize the global thread tree.
3. This defines a new logical instant $t_{k+1} = t_k + d$. Restart 1.

2.1 Multiclock Services

Several instructions explicitly refer to duration values. As explained in Section 1.1.1, durations values can be expressed with different time units, corresponding to different clocks. We assume that these clocks are managed in an external module (called *clocks module*) accessible through services described as follows.

- it is possible to be notified at any time of the current date in any time unit (it is needed for dealing with some reserved variables in expressions).
- any two delays in same or different units are comparable (it is needed for evaluating some Boolean predicates on durations in the expressions).
- it is possible to start a *timer* attached to a node p in the global thread tree, given a delay d in a time unit tu.

  The node p will be notified of the expiration of the delay after d units of tu.

(opt) it is possible to start a *recursive timer* given a period value d in a time unit tu, a delay d' in a time unit tu', and a node p in the global thread tree.

The node p will be notified every d units of tu until the expiration of the delay d'.

The notifications of expiration are considered in the same way as external events in Section 2.4. The recursive timers are used to represent directly Antescofo's periodic loops with an expiration date.

2.2 States

A *local store* is a mapping from a finite subset of $\mathcal{X}_l$ into values. Given a local store σ, $x \in \mathcal{X}_l$ and a value v, we write $\sigma[x \mapsto v]$ the store σ' defined by $dom(\sigma') = dom(\sigma) \cup \{x\}$ and $\sigma'(x) = v$ and $\sigma'(y) = \sigma(y)$ for all $y \in dom(\sigma) \setminus \{x\}$.
A *global store* is a mapping from a finite subset of $\mathcal{X}_g$ into values and from the finite set of signals occurring in $\mathcal{M}$ into Boolean values. The latter part is used to accumulate signals sent during the execution of synchronous instructions.

We shall use a similar notation for global stores and local stores. By abuse of notation, we make no distinction between a store and his homomorphic extension to expressions.
A *local state* is a pair denoted $\langle \ell, \sigma \rangle$ where ℓ is a location instruction and σ is a local store. It is called synchronous when $\mathcal{M}(\ell)$ is a synchronous instruction, and asynchronous when $\mathcal{M}(\ell)$ is an asynchronous instruction.
A concurrent state expression T, or *tree* for short, is either a local state or one of TRUE, FALSE, ERROR, AND(T_1, T_2), XOR(T_1, T_2), SOR(T_1, T_2), where T_1 and T_2 are trees. The operators AND and XOR are associative and commutative (not SOR). We use the notation $C[T_1]$ to denote a tree made of a context C and a subtree T_1. The evaluation of the trees is defined in Section 2.4.
The *global state* is a pair $\langle \gamma, T \rangle$ where γ is a global store and T is a tree called *global tree*.

2.3 Synchronous Transitions

A synchronous transition between global states represent a maximal execution of successive synchronous instructions, until the global state contains only asynchronous instructions. The synchronous instructions are executed sequentially, following the ordering $\ll$. The signals sent during the execution of synchronous instructions are accumulated in the global store.

We define synchronous transitions with a small step semantics, based on a binary relation, denoted $\rightarrow$, on global states, representing the execution of one synchronous instruction. Let $g = \langle \gamma, C[\langle \ell, \sigma \rangle] \rangle$ be a global state. We define the relation $\rightarrow$ according to the case of $\mathcal{M}(\ell)$.

if $\mathcal{M}(\ell) =$ `emit` s then $g \rightarrow \langle \gamma[s \mapsto true], C[\langle \ell + 1, \sigma \rangle] \rangle$

if $\mathcal{M}(\ell) =$ `send` a, then $g \rightarrow \langle \gamma, C[\langle \ell + 1, \sigma \rangle] \rangle$

if $\mathcal{M}(\ell) = x := e$, x is local and $\gamma(\sigma(e))$ evaluates to v, then $g \rightarrow \langle \gamma, C[\langle \ell + 1, \sigma[x \mapsto v] \rangle] \rangle$

if $\mathcal{M}(\ell) = x := e$, x is global and $\gamma(\sigma(e))$ evaluates to v, then $g \rightarrow \langle \gamma[x \mapsto v], C[\langle \ell + 1, \sigma \rangle] \rangle$

if $\mathcal{M}(\ell) =$ `stop`, then $g \rightarrow \langle \gamma, C[\text{TRUE}] \rangle$

if $\mathcal{M}(\ell) =$ `if` e `jump` ℓ', and $\gamma(\sigma(e))$ evaluates to *true*, then $g \rightarrow \langle \gamma, C[\langle \ell', \sigma \rangle] \rangle$

if $\mathcal{M}(\ell) =$ `if` e `jump` ℓ', and $\gamma(\sigma(e))$ evaluates to *false*, then $g \rightarrow \langle \gamma, C[\langle \ell + 1, \sigma \rangle] \rangle$

if $\mathcal{M}(\ell) =$ `spawn` ℓ', then $g \rightarrow \langle \gamma, C[\text{AND}(\langle \ell + 1, \sigma \rangle, \langle \ell', \sigma \rangle)] \rangle$

if $\mathcal{M}(\ell) =$ `spawn`$_0$ ℓ', then $g \rightarrow \langle \gamma, C[\text{AND}(\langle \ell + 1, \sigma \rangle, \langle \ell', \varnothing \rangle)] \rangle$

if $\mathcal{M}(\ell) =$ `asap` $\ell_1 \ldots \ell_n$, then $g \rightarrow \langle \gamma, C[\text{XOR}(\langle \ell_1, \sigma \rangle, \ldots, \langle \ell_n, \sigma \rangle)] \rangle$

if $\mathcal{M}(\ell) =$ `sustain` ℓ_1 `jump` ℓ_2, then $g \rightarrow \langle \gamma, C[\text{SOR}(\langle \ell_1, \sigma \rangle, \langle \ell_2, \sigma \rangle)] \rangle$

Some cases that require $\gamma(\sigma(e))$ to be evaluable. If this condition is not met, then the node $\langle \ell, \sigma \rangle$ is reduced to ERROR.
Moreover, we assume that the clock module is called when entering, from g, a local state $\langle \ell, \sigma \rangle$ at node p of the global tree, in the following cases:

when $\mathcal{M}(\ell) = \texttt{await}\, e\, \texttt{jump}\, \ell'$, if $\gamma(\sigma(e))$ evaluates to a delay value d, then, if $d > 0$, start a timer with d and p. Otherwise, the whole global tree reduces to ERROR.

when $\mathcal{M}(\ell) = \texttt{repeat}\, e\, \texttt{jump}\, \ell'\, \texttt{for}\, e'$, if $\gamma(\sigma(e))$ and $\gamma(\sigma(e'))$ evaluate respectively to delay values $d > 0$ and d', then, if $d' > 0$, start a recursive timer with d, d' and p. Otherwise, the whole global tree reduces to ERROR.

The reflexive-transitive closure of $\rightarrow$ is denoted $\xrightarrow{*}$, and the operator $\downarrow_*$ of normalization by $\rightarrow$ is defined by (using postfix notation): $g' = g \downarrow_*$ iff $g \xrightarrow{*} g'$ and for all g'' such that $g' \xrightarrow{*} g''$ then $g'' = g'$. Note that if $g' = g \downarrow_*$ then all the local states occurring in g' are asynchronous.

2.4 Asynchronous Transitions

We define now asynchronous transitions between global states. For this purpose we use the notion of *logical event*, denoted τ..., which is one of

- ε, representing an internal event,
- a symbol g representing a notification of the expiration of a delay to a node p in the global tree,

(opt) the symbol $\mathsf{step}\, p$ representing a notification of the expiration of a recursive delay to a node p in the global tree,

- an input symbol $i \in \mathcal{I}$, representing the recognition of i,
- a global store α of the form $\{x \mapsto v\}$, representing the assignment of the global variable x by the external environment.

Each of them represent an event which can unlock asynchronous instructions, and will be used to define our time model in Section 2.5.
We first define relations $\xrightarrow{\tau,\gamma}$ between local states indexed by a logical event τ and a global store γ. In some cases, the top symbol in the right-hand-side is marked (underlined) to indicate that it has been evaluated. This marking will be used below for the definition of further transformations for XOR, SOR, AND.

if $\mathcal{M}(\ell) = \texttt{await}\, e\, \texttt{jump}\, \ell'$, then

$\langle \ell, \sigma \rangle \xrightarrow{\varepsilon,\gamma}$ ERROR if $\gamma(\sigma(e))$ does not evaluate to a delay value

$\langle \ell, \sigma \rangle \xrightarrow{\varepsilon,\gamma} \underline{\langle \ell', \sigma \rangle}$ if $\gamma(\sigma(e))$ evaluates to a delay 0

$\langle \ell, \sigma \rangle \xrightarrow{\mathsf{done}\, p,\gamma} \underline{\langle \ell', \sigma \rangle}$ if the local state $\langle \ell, \sigma \rangle$ occurs at node p in the global tree

if $\mathcal{M}(\ell) = \texttt{repeat}\, e\, \texttt{jump}\, \ell'\, \texttt{for}\, e'$, then

$\langle \ell, \sigma \rangle \xrightarrow{\varepsilon,\gamma}$ ERROR if $\gamma(\sigma(e))$ or $\gamma(\sigma(e'))$ does not evaluate to a delay value, or if $\gamma(\sigma(e))$ evaluates to a delay 0

$\langle \ell, \sigma \rangle \xrightarrow{\varepsilon,\gamma} \underline{\langle \ell', \sigma \rangle}$ if $\gamma(\sigma(e'))$ evaluates to a delay 0

$\langle \ell, \sigma \rangle \xrightarrow{\mathsf{step}\, p,\gamma} \underline{\text{AND}}(\langle \ell, \sigma \rangle, \langle \ell', \varnothing \rangle)$ if the local state $\langle \ell, \sigma \rangle$ occurs at node p in the global tree

if $\mathcal{M}(\ell) =$ `receive` i `jump` ℓ', then

$\langle \ell, \sigma \rangle \xrightarrow{i,\gamma} \underline{\langle \ell', \sigma \rangle}$

if $\mathcal{M}(\ell) =$ `present` s `jump` ℓ', then

$\langle \ell, \sigma \rangle \xrightarrow{\varepsilon,\gamma} \underline{\langle \ell', \sigma \rangle}$ if $\gamma(s) = \mathit{true}$

if $\mathcal{M}(\ell) =$ `suspend` e `jump` ℓ', then

$\langle \ell, \sigma \rangle \xrightarrow{\varepsilon,\gamma} \underline{\langle \ell', \sigma \rangle}$ if $\gamma(\sigma(e))$ evaluates to *true*

$\langle \ell, \sigma \rangle \xrightarrow{\alpha,\gamma} \underline{\langle \ell', \sigma \rangle}$ if α is a global store and $\alpha(\gamma(\sigma(e)))$ evaluates to *true*

We define below another set of transformation rules for trees called normalization rules. In these rules, T, $\underline{T}$, $\underline{T'}$, X represent trees that cannot be transformed anymore (normal forms). Moreover, the top symbol of $\underline{T}$, $\underline{T'}$ is marked, the top symbol of T is unmarked, and the top symbol of X is either marked or unmarked. Note in particular that $\underline{T}$ cannot be ERROR. Remember that XOR and AND are associative and commutative.

$\text{XOR}(\underline{T}, T) \to \underline{T}$

$\text{XOR}(\underline{T}, \underline{T'}) \to \text{ERROR}$

$\text{XOR}(X, \text{ERROR}) \to \text{ERROR}$

$\text{SOR}(T, \underline{T}) \to \underline{T}$

$\text{SOR}(\underline{T}, \underline{T'}) \to \text{ERROR}$

$\text{SOR}(X, \text{ERROR}) \to \text{ERROR}$, $\text{SOR}(\text{ERROR}, X) \to \text{ERROR}$

$\text{AND}(X, \text{TRUE}) \to X$

$\text{AND}(X, \text{ERROR}) \to \text{ERROR}$

We denote $T \downarrow_{\tau,\gamma}$ the tree T' obtained from T in three steps:

1. application of the rules $\xrightarrow{\tau,\gamma}$ at most once to each leaf of T. When $\xrightarrow{\tau,\gamma}$ is appliable to one leaf at least, then we say that the logical event τ *unlocks* T *wrt* the global store γ.

2. iterated application of the normalization rules to internal nodes, as long as possible

3. finally, removing of the marks (i.e. $\underline{\langle \ell', \sigma \rangle}$ is renamed into $\langle \ell', \sigma \rangle$, $\underline{\text{AND}}$ is renamed into AND etc).

2.5 Execution

The execution of a machine $\mathcal{M}$ is a sequence of global states, each of them being obtained from the previous one in two steps: one synchronous transition (defined in Section 2.3 as a maximal sequence of execution of synchronous instructions), followed by one asynchronous transition (defined in Section 2.4 as the parallel and simultaneous execution of asynchronous transitions). The dates of appearance of each global state (the beginning of execution of synchronous transitions) will be called *logical instants*. They correspond to the dates of logical events described at the beginning of Section 2.4. Following the time model of Antescofo (see § 3 of [5]). Hence every new logical instant correspond to one of: the expiration of a delay, the recognition of an input event or internal signal, the assignment of a global variable by the external environment.

Formally, let us define the first logical instant as $t_0 = 0$ and assume an initial global state g_0 of the form $g_0 = \langle \varnothing, T_0 \rangle$, where the initial global tree T_0 has one single node labeled by $\langle 0, \varnothing \rangle$ (0 is the first location of $\mathcal{M}$). The rest of the sequences of logical instants and global states is defined recursively as follows.

Given a global state $g_k = \langle \gamma_k, T_k \rangle$ at logical time $t_k \geq 0$, with $k \geq 0$, let $g'_k = \langle \gamma'_k, T'_k \rangle = g_k \downarrow_*$. The next logical instant t_{k+1} and global state g_{k+1} are defined as follows.

If ε unlocks T'_k wrt γ'_k, then $t_{k+1} = t_k$, and $g_{k+1} = \langle \gamma_{k+1}, T_{k+1} \rangle$ where $T_{k+1} = T'_k \downarrow_{\varepsilon,\gamma'_k}$ and $\gamma_{k+1} = \gamma'_k$.
Note in particular that the signals are not reset in γ_{k+1}.

Otherwise, $t_{k+1} > t_k$ is the date of the next logical event τ, which can be one of

(i) $\mathsf{done}\, p$,

(ii) $\mathsf{step}\, p$,

(iii) $i \in \mathcal{I}$,

(iv) a global store $\alpha = \{x \mapsto v\}$.

Let $g_{k+1} = \langle \gamma_{k+1}, T_{k+1} \rangle$ where γ_{k+1} is obtained from $\alpha \circ \gamma'_k$ by resetting every signal assignment to *false* and $T_{k+1} = T'_k \downarrow_{\tau,\gamma'_k}$.

The execution depends on the behavior of the environment but it is deterministic in the sense that the same behavior givens the same execution of $\mathcal{M}$. Observational behavior can be characterized by the timed trace containing the input symbols received with `receive`, the global variables modified by the environment and the output symbols emitted with `send`, each with the corresponding logical instant.

3 Implementation Issues

3.1 Clock Services

The clock services can be implemented using one ordered queue of delays for each clock.

3.2 Time Safety

The above definition of execution is theoretical and assumes that the synchronous transition take zero delay. In reality, we have to take care of the time needed to do these transitions. Moreover, handling the events that define logical instants, and the reorganization of the global tree, are assume instantaneous in Section 2, we also need to take care of the time needed to perform these

task in reality. Since there is no control on the environment these issues can not always be solved, let us discuss in this paragraph a best effort strategy to addresses them.

Let g_k be a global state, reached at the logical instant t_k (as defined in Section 2.5). Let δ_k be the time needed to perform the synchronous transition and compute $g'_k = g_k \downarrow_*$ and let ϵ_k be the time needed for handling i and making the synchronous transition from g'_k to g_{k+1}. For convenience, we let $\epsilon_{-1} = 0$. Let $\theta_k = \delta_k + \epsilon_{k-1}$ for $k \geq 0$.
Let us assume that the theoretical delay $d_k = t_{k+1} - t_k$, as defined in Section 2.5, corresponds to the arrival of an event i (case (iii)). If $d_k \geq \theta_k$, then time safety is ensured. This can be depicted as follows, with the time flowing from left to right.

g_k	g'_k	i	g_{k+1} ready
$t_k + \epsilon_{k-1}$	$t_k + \theta_k$	$t_k + d_k = t_{k+1}$	$t_{k+1} + \epsilon_k$

If $d_k < \theta_k$, then there is a difference between logical time and real time that must be handled.

g_k	i	g'_k	g_{k+1} ready
$t_k + \epsilon_{k-1}$	$t_k + d_k = t_{k+1}$	$t_k + \theta_k$	$t_k + \theta_k + \epsilon_k$

For instance, the difference $d_k - \theta_k$ can be retrieved from the delay of a `await` instruction occurring next to g'_k. But we cannot guarantee that it is always possible.

3.3 Static Analysis

A strategy to predict statically time safety could be to use estimation of worst case execution time (WCET) of the possible sequences of synchronous instruction in $\mathcal{M}$. Note that these values depend on the execution platform. Knowing on these durations, the analysis would then consist in estimating whether the durations in the asynchronous `wait` instructions are compatible with the WCETs. Moreover, one has to deal with the unpredictable timing for external events. One approach could be to infer a linear constraint on these timings for ensuring there compatibility with WCETs. An alternative is to solve a 2 players safety game on the graph defined by the global states of $\mathcal{M}$, extended with the timing information.

The above approaches are similar to techniques used in the compilation of (X)Giotto into Ecode [3, 7]. There are some differences however. First, in Ecode, the analogous of the above synchronous instructions is written in a conventional programming language like C, for which procedures for estimation of WCETs exist. Second, all the timings in Giotto are expressed in milli-seconds, whereas timings can be expressed in multiple clocks in Antescofo.

Another interesting question in this setting is whether the structure of the intermediate code obtained from Antescofo programs is sufficiently simple in order to avoid an exponential explosion in a time safety analysis.

Note that the execution of synchronous instructions following the ordering $\ll$ and the global execution scheme (decomposed into synchronous and asynchronous step) permit to avoid race conditions and ensures determinism.

References

[1] J. Echeveste, A. Cont, J.-L. Giavitto, and F. Jacquemard. Operational semantics of a domain specific language for real time musician-computer interaction. *Discrete Event Dynamic Systems*, 23(4):343–383, Aug. 2013.

[2] J. Echeveste, J.-L. Giavitto, and A. Cont. A Dynamic Timed-Language for Computer-Human Musical Interaction. Research Report RR-8422, INRIA, Dec. 2013.

[3] A. Ghosal, T. A. Henzinger, C. M. Kirsch, and M. A. Sanvido. Event-driven programming with logical execution times. In *International Workshop on Hybrid Systems: Computation and Control (HSCC)*, volume 2993 of *LNCS*, pages 357–361. Springer, 2004.

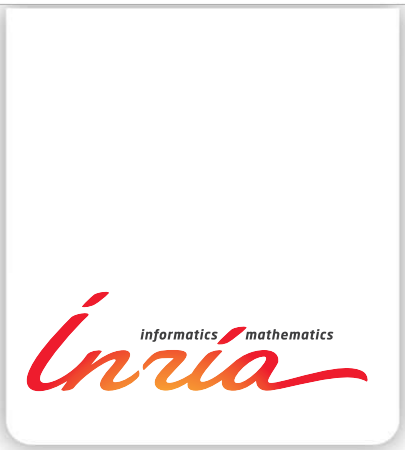
[4] A. Ghosal, A. Sangiovanni-Vincentelli, C. M. Kirsch, T. A. Henzinger, and D. Iercan. A hierarchical coordination language for interacting real-time tasks. In *Proceedings of the 6th ACM &Amp; IEEE International Conference on Embedded Software*, EMSOFT '06, pages 132–141, New York, NY, USA, 2006. ACM.

[5] J.-L. Giavitto. Antescofo: a quick introduction to version 0.51. Technical report, IRCAM UMR STMS 9912 – CNRS – UPMC – INRIA/MuTant, November 2013.

[6] D. Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, June 1987.

[7] T. A. Henzinger and C. M. Kirsch. The embedded machine: Predictable, portable real-time code. *ACM Trans. Program. Lang. Syst.*, 29(6), Oct. 2007.

[8] E. A. Lee, S. Neuendorffer, and G. Zhou. Dataflow. In C. Ptolemaeus, editor, *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014.



RESEARCH CENTRE
PARIS – ROCQUENCOURT

Domaine de Voluceau, - Rocquencourt
B.P. 105 - 78153 Le Chesnay Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399