Multimedia scheduling for interactive multimedia systems

Pierre Donat-Bouillud

MASTER RESEARCH INTERNSHIP

MASTER THESIS

Multimedia scheduling for interactive multimedia systems

Domain : Multimedia, programming language and embedded systems

Author:
Pierre DONAT-BOUILLUD

Supervisors:
Arshia CONT
Florent JACQUEMARD
Mutant team - Inria

Abstract

Scheduling for real-time interactive multimedia systems (IMS) raises specific challenges that require particular attention. Examples are triggering and coordination of heterogeneous tasks, especially for IMS that use a physical time, but also a musical time that depends on a particular performance, and how tasks that deal with audio processing interact with control tasks. Moreover, IMS have to ensure a timed scenario, for instance specified in an augmented musical score, and current IMS do not deal with their reliability and predictability.

We present how to formally interleave audio processing with control by using *buffer types* that represent audio buffers and the way of interrupting computations that occur on them, and how to check the property of *time-safety* of IMS timed scenarios, in particular augmented scores for the IMS Antescofo for automatic accompaniment developed at Ircam. Our approach is based on the extension of an intermediate representation similar to the E code of the real-time embedded programming language Giotto, and on static analysis procedures run on the graph of the intermediate representation.

Acknowledgements

I would like to thank my supervisors Arshia Cont and Florent Jacquemard for their supervision, their advice to solve the numerous obstacles I was facing on this path of adding a small building block to science, and for reviewing this report. I would also like to thank José Echeveste, Clément Poncelet, and Pierrick Lebraly, for reviewing this report and making a huge number of remarks, and also Jean-Louis Giavitto, for reviewing but also for the multiple conversations we have had about various subjects of computer science. I also thank Nicolás Schmidt Gubbins, Philippe Cuvillier and Grig Burloiu for the discussions, about computer science as well as other very enlightening topics!... Finally, I would like to thank my parents to have accommodated me in Paris.

Contents

Introduction

Embedded systems are at work in our daily life: they drive cars, command nuclear plants, are used in critical systems such as in airplanes. However, they are not as visible as modern personal computers such as in connected objects which are taking over society. Embedded systems are asked more and more to interact continuously with their environments and are tightly linked to sensors capturing the physical world. This evolution has led to a new trend in research and development commonly referred to as Cyber Physical Systems [24]. CPS are highly distinguished by their need to interact in real-time with events gathered by sensors and by their time-awareness.

Interactive multimedia systems (IMS) [38] combine audio and video processing (but also light, movement...) with ways of controlling the output of audio and video by reacting to events, which can range from the move of a slider or periodic events sampled on a curve to the detection of a note in an audio stream: they accept inputs such as sounds, MIDI[1] events, OSC[2] messages, and can react by processing sounds (sound transformation or spatialization), triggering processes, changing parameters; therefore they combine control event and signal processing, and can be also qualified as *cyberphysical systems*. There is also a growing trend in IMS for porting them to credit card-sized single-board computers such as Udoo[3] or Raspberry Pi.[4]

Scheduling in interactive multimedia systems, as they are reactive and real-time systems, entails several challenges such as how to deal with the perceptive time limits of human beings,[5] how to schedule digital audio and video processing, which is periodic, and control, which is mainly aperiodic, and more generally, how to take into account various rates[6] for audio and video. These IMS often involve heterogeneous models of computations, as, apart from coordinating various sound processing tasks, they can use spectral processing, physical modelling of shapes or acoustical models, transactional processing for querying data in multimedia databases and so on.

In this study, we will focus on the IMS that mainly deal with audio. The video constraints regarding real-time are less stringent: studies have shown that in multimedia setups, correctness for real-time audio is more stringent that video in general because human ear is more sensitive to audio gaps and glitches that human eye is to video jitter [41].

IMS have strong requirements of reliability and predictability, in particular value-determinism and time-determinism, while interacting in a heterogeneous and unpredictable physical environment. To address these issues, IMS have gathered contributions from fields such as machine listening, signal processing, real-time systems and programming languages.

Currently, IMS are statically scheduled: priority is given to audio computing over control event processing, which can be restrictive in some applications such as automatic accompaniment and automatic improvisation. There is a consensus among users that leveraging IMS to dynamic scheduling approaches is critical. Another drawback is weak resilience and predictability of IMS, in particular regarding timings. Thus, formal analysis procedures must be investigated.

Our goal in this work is to tackle scheduling for IMS at several scales: the scale of reaction and coordination of the tasks of the IMS to check their schedulability while ignoring the nature of

[1] Musical instrument digital interface: protocol for communication for music devices and computers.

[2] Open Sound Control: recent protocol to connect music devices and computers. `http://opensoundcontrol.org/`.

[3] `http://udoo.org/`

[4] `https://www.raspberrypi.org/`

[5] Given that two events are perceived to be not simultaneous if there are more than 20 ms between them, how achieving real-time scheduling?

[6] Samplerate for audio, for instance, 44.1 kHz, and image rate for video, for example, 35 fps.

these tasks, and to zoom on a specific kind of tasks, that's to say the signal processing tasks and how data flows among them.

In this report, we review real-time scheduling and some IMS in part I, and in particular how IMSs deal with the interaction between control and audio processing. Antescofo [6, 10], a score-following system, couples an artificial listening machine and a domain-specific real-time programming language which is the augmented analogue of a score, in order to have a computer play synchronously with human musicians and act like an artificial musician. It is mainly used for mixed music: a music which mixes acoustic instruments and electronic sounds or effects. We present more thoroughly the language in part II as well as sketching a new multirate and typed audio processing infrastructure for Antescofo, which is formalized using a *typed buffer flow*. In part III, after considering the main paradigms for real-time programming, we extend an intermediate language [33] for Antescofo inspired by E code [18] by adding tasks to it to perform static analysis on the graph representation of this intermediate code, in order to test schedulability, and derive bounds on *tempi* for a score.

Part I
Multimedia scheduling

Interactive multimedia systems are real-time systems; as they deal with audio and video streams, they have to provide audio or video frames periodically before a precise deadline. We will first present the main concepts used to tackle real-time programming, and then, how scheduling works in some IMS.

Interactive multimedia systems are softwares that are part of an operating system, and the scheduling is carried out in several layers (see Fig. 1), the multimedia software, then a multimedia library, and ultimately, the OS scheduler.

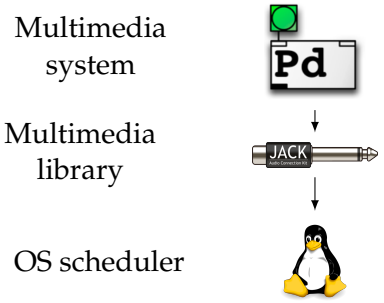


Figure 1: Scheduling is organized within several layers.

1 Real-time scheduling

Here, we present several principles to deal with real-time scheduling, some important algorithms, how to handle both periodic and aperiodic actions and events, which is particularly crucial for IMS which have to react input from the users, and overload, from the multimedia system as well as from the surrounding operating systems, since multimedia systems are most of the times deployed on mainstream operating systems which do not facilitate hard real-time programming.

1.1 General characteristics

Real-time scheduling can be characterised [7] by their behaviour into several categories:

Dedicated to hard or soft real-time. We distinguish between *hard real-time* and *soft real-time*. *Hard real-time* tasks are tasks for which their deadline must absolutely be respected, because for instance not respecting it would lead to a crash of a drone. On the contrary, a *soft real-time* task is a task for which respecting the deadline is preferable but exceeding the deadline only entail a degraded quality of service, for instance, clicks in an audio system, or late delivery in a communication system.

Static or dynamic scheduling (also called online and offline scheduling) Static scheduling prepares a scheduling sequence for all tasks before the launch of the tasks before starting execution of the system. It requires to know in advance the tasks that are going to be executed, and their arrival and release time. It only induces a low overhead because the scheduling schema is simply written in a table read at runtime.

Dynamic scheduling has more overheard, and is less precise than the offline approach, but can deal with the unpredictability of sporadic or aperiodic tasks, for instance.

Preemptive and not preemptive. When tasks can be stopped in the middle of a computation, in order to launch other tasks, and can be continued later: they are called *preemptible* tasks. On the other hand, some tasks, such as interrupt handlers, must not be preempted and are *non-preemptive* tasks (also called *immediate tasks*).

Dependencies or not between tasks. Real-time tasks can depend on each other, that's to say, one task needs the output of another task to start execution. This is the case in digital signal processing graphs for instance, where the audio signal flows from one node to others and is processed through effects.

Periodic tasks or sporadic tasks. A task can be *periodic*, executed with period T, or *aperiodic* (GUI events, for instance), or *sporadic*, that's to say tasks that regularly repeat but without a fixed period.

Utilization and overload. Utilization defines how much (as a ratio) of the processor is used by the tasks that are executed. If the ratio is more than one, it means that there is *overload*.

Latency is time between input and output for a processed piece of data.

Jitter is the variation of latency (allowed before deadline but not after).

Reaction to permanent or transient overhead. A *transient overload*, contrary to a *permanent overload*, is a burst in utilization of the processor, when the utilization is more than 1.

1.2 Common approaches

We introduce common approaches to real-time scheduling deployed in majority of systems and present the concept of schedulability test to check whether tasks can be scheduled with a given scheduling algorithm.

1.2.1 Rate Monotonic (RM)

RM [28] is usually used for tasks that execute periodically, without dependencies. It is optimal with respect to feasibility: if it exists a priority assignment that yields a feasible schedule, then RM yields a feasible schedule. It is online and uses fixed priorities.

Given a list of tasks $(T_1, \ldots, T_n)$ and periods $(p_1, \ldots, p_n)$, T_i has to execute exactly once during each period for each interval p_i. Hence, the deadline for the j-th execution of T_i is $r_{i,1} + jp_i$ where

$r_{i,1}$ is the release time[7] of the first execution of T_i. The principle of RM is to give higher priority to tasks with smaller periods.

A sufficient schedulability test is:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n \times (2^{\frac{1}{n}} - 1)$$

where T_i are the periods of the tasks and C_i their worst case execution time.

When the number of tasks tends to infinity, U tends to a maximum theoretical utilization of 88%. As this schedulability test is a sufficient condition, and not a necessary one, there may exist in some cases scheduling schemes that achieve a better utilization that RM.

1.2.2 Earliest deadline first (EDF)

EDF is an online scheduling algorithm, with dynamic priorities (the same task can have various priorities after several activations). In the EDF policy, tasks with the nearest absolute deadlines are scheduled first, that's to say, priorities are the inverse of deadlines. This policy is optimal, preemptive, and is used for periodic or sporadic tasks, without dependencies. EDF minimizes the minimal lateness and is optimal for feasibility.

A sufficient schedulability test for EDF exists when deadling with periodic tasks:

$$U = \sum_{i=1}^{n} \frac{C_i}{D_i} < 1$$

where D_i are the deadlines of the tasks and C_i, their worst case execution times. Contrary to RM, EDF can ensure a 100% utilization.

If there are only periodic tasks (thus, deadlines are equal to periods), the condition is also a necessary condition.

Tasks can also depend on other tasks: they must not be executed if another task has not finished. However, EDF with precedences is not optimal. In EDF* by Chetto [5], deadlines are modified as follows:

$$d'_i = min(d_i, min_{j \in D(i)}(d'_j - e_j))$$

where $D(i)$ is the set of tasks that depend on task i, d_i, its deadline, e_j, its execution time, and d'_i, the modified deadline.

1.3 Dealing with sporadic and aperiodic events

In a system with periodic and aperiodic tasks, periodic tasks can interfere with aperiodic tasks, and make them execute too late. There are three main ways of dealing with this interaction between aperiodic and periodic tasks [4]:

Background scheduling. In this approach, aperiodic tasks are scheduled when no periodic tasks are ready to execute. If there are several aperiodic tasks, they are executed in FIFO order. A disadvantage is the huge response time.

[7] Not to be confused with start time: start time is when the task is given to the scheduler, and release time, when the scheduler actually launches the task.

Server. A *server* is a periodic task the purpose of which is to serve aperiodic requests. A *server* has a period and a granted computation time (*server capacity* or *budget*) that it can use before letting other periodic tasks resume execution. In some case, servers can degrade utilization.

Slack stealing and joint scheduling. It consists of stealing processing time from the periodic tasks without having them miss deadlines by using *laxity*, *i.e.* the temporal difference between the deadline, the ready time and the run time.

1.4 Overload

In case of (unpredictible) overload, several policies [42] can be used. There are two main strategies: having several versions of a task that don not yield the same quality, or a way of cancelling tasks over other tasks.

For the first policy, there can be a primary program P_1, and an alternate program P_2; P_1 has a good quality of service and takes an unknown duration to complete, on the opposite P_2 is deterministic, has a bad quality of service, but has a known worst case execution time. Two strategies can be used to choose between the two tasks, *first chance* and *last chance*, depending on whether the good quality tasks is stopped to launch P_2, or if P_2 has terminated and there is still time, P_1 is executed while the results of P_2 are kept in case of P_1 exceeding its deadline.

Another method is the *imprecise computation model*: multimedia processing is separated into a mandatory part, and an optional part that can be dropped.

For the second policy, tasks are given an *importance* value and in case of overload, tasks with the least *importance* are killed until there is no more overload. To handle the termination of such tasks gracefully, tasks present two modes, a normal mode, and a survival mode[8], with an abortion or an adjournment property (such that freeing memory, or saving partial computations to continue later).

2 Scheduling in interactive multimedia systems

Interactive multimedia systems deal with audio and video streams, where a buffer has to be filled periodically, and with controls that can be aperiodic (a GUI change) or periodic (a low frequency oscillator). The audio and video streams can use different samplerates, and the controls, other rates, that have to be accommodated together, with several priorities. Moreover, audio streams and controls are connected in a *dataflow graph*, which can also involve spectral processing, granular synthesis using a corpus of sounds stored in a database. How articulating control and processing is one of the challenges of scheduling IMS: control can take too much time and delay audio processing; audio processing often processes by chunks to use the vector instructions of the targeted processor, so that control cannot occur inside such a chunk. We will see that in the following IMS, control indeed happen at the boundaries of a buffer, which entail that control is not sample-accurate.

[8] It can remind exception handlers.

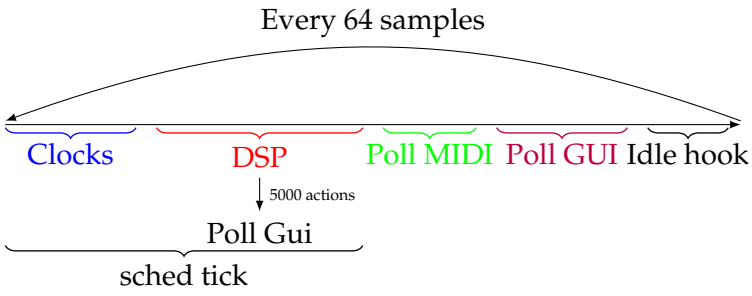


Figure 2: Scheduling cycle in Puredata (polling scheduler)

2.1 Patching as graphical programming environment

The Patcher family, with Max/MSP [45], PureData [35], jMax [8], MAX/FTS [37] originally developed by Miller Puckette at Ircam [36], emphasises visual programming for dataflow processing. Functions are represented by boxes, placed on a screen called canvas. Objects are connected together with links, and data flows from one object to another through these links. Here, we present more specifically PureData, which is opensource.

In Puredata, actions can be timestamped, for instance the boxes `metro`, `delay`, `line`, or `timer`. GUI updates are also timestamped. Other incoming events are MIDI events, with `notein` and OSC. The scheduling in Puredata is block-synchronous, that's to say controls occurs at boundaries (at the beginning) of an audio processing on a block of 64 samples, at a given samplerate. For an usual samplerate of 44.1 kHz, a scheduling tick lasts for 1.45 ms. Depending on the audio backend,[9] the scheduler can be a polling scheduler (see Fig. 2) or use the audio callback of the backend.

Samplerate and block size can be changed locally in a patch using the `block~` box.

2.2 SuperCollider

SuperCollider [2] is an interactive multimedia system mainly used for audio synthesis and for live coding.[10] SuperCollider uses a client/server architecture (see Fig. 3): on the client side, a Smalltalk-like language makes it possible to compose the piece, and generates OSC messages which are sent to the server. OSC messages can be timestampped and in this case, they are put in a priority queue and executed on time, or can be sent without a timestamp and are then processed when received. All events such that their timestamps is lower than the tick size (or scheduling period) are executed at once.

The server uses three threads, a network thread, a real-time thread for signal processing, and a non-real time thread for purposes such as reading files. Commands among threads are communicated via lock-free FIFOs.

There are three clocks in the language, `AppClock`, a clock for non-critical events that is allowed to drift, `SystemClock` for more precise timing, and not allowed to drift, and `TempoClock`, the ticks

[9] Jack or ALSA on Linux, CoreAudio on MAC, or WASAPI on Windows, for example.

[10] Live coding consists of creating a musical piece at the time of the performance by coding it in a DSL in front of the attendance: the editing of the lines of codes is usually projected as the musical piece is created.

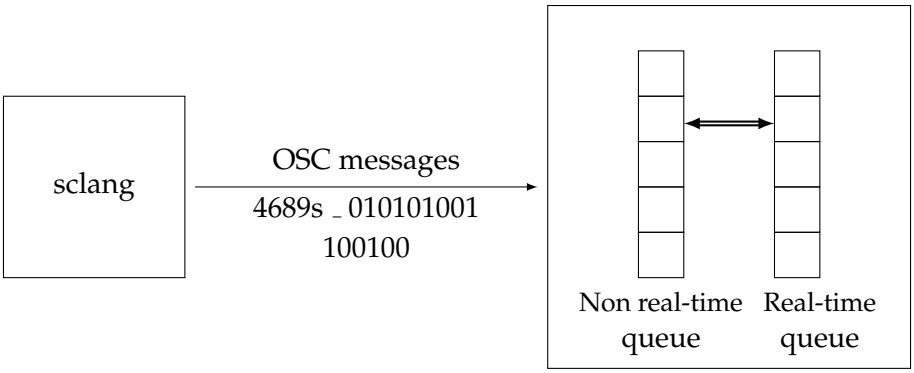


Figure 3: Client-server architecture of SuperCollider.

of which are beats synchronized with the actual tempo. As the tempo changes, the actual next event date can be recomputed.

2.3 ChucK

Chuck [44] is a *strongly timed* (*i.e.* with an explicit manipulation of time, thanks to variable `now`), synchronous, concurrent, and *on-the-fly* music programming language.. It is mainly used for live-coding. It is compiled to bytecode and executed in a virtual machine.

Time and event-based programming mechanism. Time can be represented as a duration (type `dur`) or a date (type `time`). The keyword `now` makes it possible to read the current time, and to go forward in time. Time can also be advanced until an event, such as a MIDI event, an OSC event, or a custom event, is triggered.

```
1::second => now;//advance time by 1 second

now + 4::hour => time later;
later => now;//Advance time to date now + 4 hours

Event e;
e => now;//Wait on event e
```

`now` allows time to be controlled at any desired rate, such as musical rate (beats), as well as sample or subsample rate, or control rate.

Chuck makes it possible to use an arbitrary control rate (control occurs after the assignation to `now` stops blocking) and a concurrent control flow of execution.

Cooperative scheduling. ChucK processes are cooperative lightweight user-spaces called *shreds* (and are scheduled by a *shreduler*); they share data and timing. When time advances thanks to `... => now`, the current *shred* is blocked until `now` attains the desired time, giving room to other *shreds* to execute.

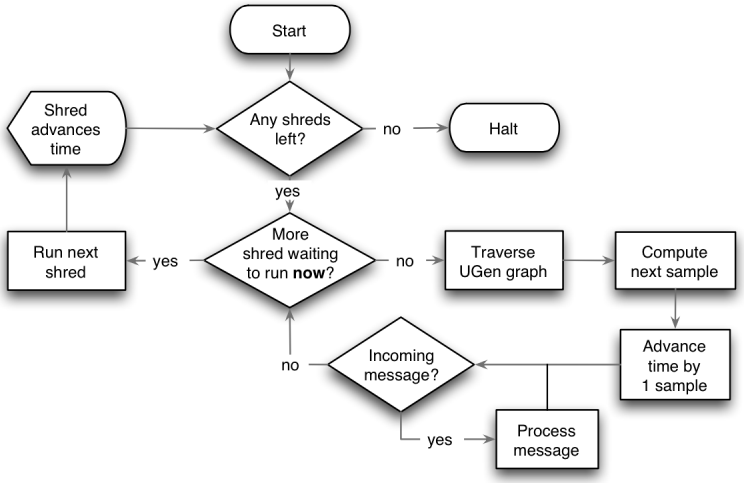


Figure 4: Shreduling in ChucK.

Shreds are synchronous (sample-based-clock of 44.1 kHz in general) and *shreduling* is deterministic: instantaneous instructions in every *shred* are executed until reaching an instruction that next advances in time.

Shreds are *shreduled* using a queue of shreds sorted by waiting time before wake up (see Fig. 4).

Schedulability. The language is type-checked, but no analysis of schedulability is performed: a very complex musical performance could well lead to lateness for some audio processing units. This absence of analysis is quite understandable when reckoning the *live coding* purpose of the language, and such problems must be dealt with by the musician at performance time.

2.4 Faust

Faust [30] is a synchronous domain-specific language dedicated to digital signal processing, which compiles to C++. It also aims at facilitating the creation of audio plugins for digital audio workstation, and thus provides a way of defining a graphical interface or to communicate to the audio process via OSC.

Audio is processed sample by sample in the language, but after compilation, it is processed on buffers (32 samples by default) and control only occurs at boundaries of the buffers.

The following example shows a Karplus-Strong model of a string coded in Faust:

```
import("music.lib");



upfront(x)  = (x-x') > 0.0;
decay(n,x)  = x - (x>0.0)/n;
release(n)  = + ~ decay(n);
```

```
trigger(n)  = upfront : release(n) : >(0.0);

size        = hslider("excitation (samples)", 128, 2, 512, 1);



dur         = hslider("duration (samples)", 128, 2, 512, 1);
att         = hslider("attenuation", 0.1, 0, 1, 0.01);
average(x)  = (x+x')/2;

resonator(d, a) = (+ : delay(4096, d-1.5)) ~ (average : *(1.0-a)) ;

process = noise * hslider("level", 0.5, 0, 1, 0.1)
: vgroup("excitator", *(button("play"): trigger(size)))
: vgroup("resonator", resonator(dur, att));
```

Special vector instructions (SSE) can then be used on the buffers. Other parallel implementations exist, with OpenMP or with a work stealing scheduler [27] (which also does pipelining).

2.5 Challenges in multimedia scheduling

Table 1 shows a comparative study of various multimedia systems. Most multimedia systems can only deal with control on boundaries of a buffer; they schedule control and audio in a round-robin fashion. Using several rates is either impossible, or difficult. No one performs any static analysis on the multimedia scenarios they make it possible to create, such as a time-safety analysis. Even if a graph of effects seems to be easily parallelisable because it makes explicit all the dependencies, few IMS are able to use several processors effectively.

Audio system	Control and audio	Several rates	Analysis	Parallelism
Patcher family	Control at boundaries of a block	`block~` in Puredata	No	Explicit: `poly~` in MAX/MSP
SuperCollider	Boundaries	Yes	No	Explicit: `Pargroup`
ChucK	Subsample accurate	Arbitrary rates	No. One must stop manually tasks in case of overload.	No
Faust	Boundaries of a block	One rate	No	Automatic
Extempore [40]	Sample	Yes *Temporal recursion*	No (live coding)	Explicit
Csound [43]	Sometimes, sample	Yes	No	Possible

Table 1: Comparison of multimedia systems with respect to scheduling.

Part II
Antescofo and audio processing

In this part, we present Antescofo, an IMS, which is embedded in a host IMS (typically PureData or Max/MSP). Antescofo harnesses the audio effect of the host environment. Our goal is to embed directly digital signal processing in Antescofo, and to formalize the interaction of control and audio computations using *buffer types*.

Antescofo [6] is an IMS for *musical score following* and *mixed music*. It is an artificial musician that can play with human musicians in real time during a performance, given an augmented score prepared by the composer. Since 2008, Antescofo has been featured in more than 40 original mixed electronic pieces by world renowned ensembles and composers, such as Pierre Boulez, Philippe Manoury, Marco Stroppa, the Radio France orchestra and Berlin Philharmonics.

Fig. 5 shows the architecture of the system: it is composed of two subsystems, a *listening machine* and a *reactive machine*. The *listening machine* processes an audio or a MIDI stream and estimates in real-time the *tempo* and the *position* of the live performers in the score, trying to conciliate performance time and score time. Position and tempo are sent to the *reactive machine* to trigger actions specified in the mixed score. These actions are emitted to an audio environment, either Max/MSP, or PureData, in which Antescofo is embedded as a performance patch. As for human musicians, Antescofo relies on adapted synchronization strategies informed by a shared knowledge of tempo and the structure of the score.

The mixed scores of Antescofo are written with a dedicated reactive and timed synchronous language, where events of the physical world – notes and rhythm played by musicians, for instance –, parts of the artificial musician, and the synchronization between them are specified. It has to take into account the temporal organization of musical objects and musical clocks (*tempi*, which can fluctuate during performance time) which are actualized in a physical, performance time. This performative dimension raises particular challenges for a real-time language:

- multiple notions of time (events and durations) and clocks (tempi and physical time)
- explicitly specifying the musical synchronisation between computer-performed and human-performed parts
- robustness to errors from musicians (wrong or missed notes) or from the listening machine (recognition error)

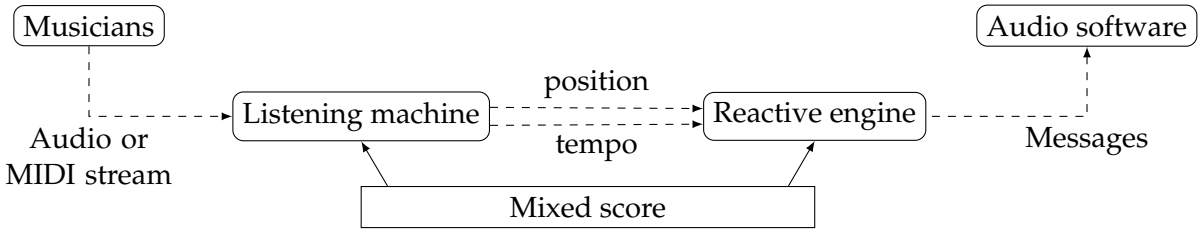
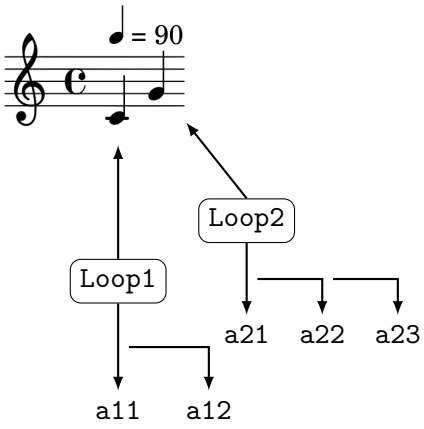


Figure 5: Architecture of the Antescofo system. Continuous arrows represent pre-treatment, and dashed ones, real-time communications.

2.6 The Antescofo language

The Antescofo language is a synchronous and timed reactive language presented for instance in [12, 10, 13].

Instrumental events The augmented score details events played by human musicians (pitches, silences, chords, trills, glissandi, improvisation boxes) and their duration (absolute, in seconds, or relative to the tempo – quaver, semiquaver...). An ideal tempo can also be given.




```
BPM 90
NOTE 60 1.0 note1
Loop Loop1 4.0
{   a11
    0.5 a12
}
NOTE 67 1.0 note2
Loop Loop2 4.0
{   a21
    0.33 a22
    0.33 a23
}
```

Figure 6: An Antescofo augmented score: actions `a11` and action `a12`, with a 0.5 delay, are associated to an instrumental event, here, a C crotchet.

Actions Actions (see Fig. 6) are triggered by an event, or follow other actions with delay. Actions follow the synchronous hypothesis, and are executed in zero time, and in the specified order.

Atomic actions It is a variable assignment, an external command sent to the audio environment, or the killing of an action.

Compound actions A `group` is useful to create polyphonic phrases: several sequential actions which share common properties of tempo, synchronization, and error handling strategies. A `loop` is similar to a `group` but its actions execute periodically. A `curve` is also similar to `group` but interpolates some parameters during the execution of the actions. Compound actions can be nested and inherit the properties of surrounding blocks unless otherwise explicitly indicated.

An action can be launched only if some condition is verified, using `guard` statements, and it can be delayed for some time after the detection of an event (or previous action). The delay is expressed in physical time (milliseconds) or relative time (beats).

Processus Processus are parametrized actions that are dynamically instantiated and performed. Calling a processus executes one instance of this processus and several processes can execute in parallel.

Expressions and variables Expression are evaluated to booleans, ints, floats, strings (atomic values) or data structures such as maps and interpolated functions (non atomic values).

Variables are either global, or declared local to a group. The assignment of a variable is an event, to which a logical timestamp is associated. `$v` is the last value of the stream of values associated with `v`, and `[date]:$v` the value at the date `date`.

`whenever` **statement** It makes it possible to launch actions if a predicate is satisfied, contrary to other actions which are linked to events sequentially notated in the augmented score.

Temporal patterns, *i.e.* a sequence of events with particular temporal constraints, can also be matched by a `whenever`.

Synchronization strategies and fault tolerance One strategy, the `loose` strategy, schedules actions according to the tempo only. Another strategy, the `tight` strategy, not only schedules according to timing positions, but also with respect to triggering relative event-positions. There are also strategies that can synchronize according to a target in the future: the tempo adapts so that the electronic part and the followed part arrive on this target at the same time.

In case of errors (missed events), two strategies are possible: if a `group` is local, the group is dismissed in the absence of its triggering event; if it is `global`, it is executed as soon as the system realizes the absence of the triggering event.

2.7 Antescofo runtime

A logical instant in Antescofo, which entails an advance in time for the reactive engine, is either the recognition of a musical event, or the external assignment by the environment of a variable, or the expiration of a delay. In one logical instant, all synchronous actions are executed in the order of their declaration.

The reactive engine maintains a list of actions to be notified upon one of those three types of events. For actions waiting for the expiration of a delay, three waiting queues are maintained: a static timing-static order queue, for actions delayed by a physical time (in seconds, for instance); a dynamic timing-static order queue, for delays related to the musician tempo (in beats); and a dynamic timing-dynamic order queue, for delays depending on local dynamic tempo expressions. Delays are reevaluated depending on changes of tempo, so that the order of actions in the queue should change. The next action to execute is the action which has the minimum waiting delay among the three queues.

2.8 *Anthèmes 2*

Listing 1 shows the beginning of *Anthèmes 2*, a mixed music piece for solo violin and electronics, which relies on Antescofo to coordinate the electronic effects and synthesis. The electronic part is still done inside Max or Puredata, but has been ported (by Nicolás Schmidt [39]) to Faust effects embedded into the Antescofo augmented part (see also section 3.1).

Anthèmes 2 uses complex control structures such as curves (in the excerpt we show, in order to increase or increase smoothly the level of volume for some effects).

```
1 Annotation "Anthemes2 Section1 READY"
2
```

```
3
4   VARIANCE 0.3
5
6   NOTE 0 1.0
7   NOTE 8100 0.1   Q7
8   Annotation "Harms open"
9   ;;;;;;;;;;;;;;;; Harmonizers
10  Curve c1
11  @grain := 0.1
12  {; bring level up to 0.8 in 25
13      $hrout
14      {
15          {0}
16          25ms {0.8}
17      }
18  }
19
20  $h1 := -1
21  $h2 := -4
22  $h3 := -8
23  $h4 := -10
24  $$linkHarm := faust::Harms($$audioIn,$h1,$h2,$h3,$h4,$hrout)
25
26  NOTE 7300 1.0
27  NOTE 7100 0.1
28  NOTE 7200 1.0
29  NOTE 6600 1/7 Q8
30  Curve c2
31  @grain := 0.1
32  {; bring level up to 0.8 in 25
33      $hrout
34      {
35          {0.8}
36          300ms {0}
37      }
38  }
39  $$linkHarm := faust::Harms($$audioIn,$h1,$h2,$h3,$h4,$hrout)
40
41  Annotation "Frequency Shifter + Delays"
42  Curve c3
43  @grain := 0.1
44  {; bring level up to 0db in 25
45      $psout
46      {
47          {0}
48          25ms {1}
49      }
50  }
51
52  $freq := -205 ; frequency shift value for fd1
53  $aux := $PITCH
```

```
$$linkFS := faust::Pitch($$audioIn, $freq, $aux, $psout)
NOTE 6900 1/7
NOTE 7200 1/7
NOTE 7300 1/7
NOTE 7400 1/7
NOTE 7500 1/7
NOTE 7700 1/7
```

Listing 1: Beginning of *Anthèmes 2* by Pierre Boulez

3 Audio processing architecture in Antescofo

Using a scheduling algorithm such as EDF as presented in part I makes it possible to handle more precisely the interaction between audio processing and control in complex scenarios: contrary to the round-robin schedulers of section 2), control and audio can be interrupted, so as to get sample-accurate control and handle control calculations that delay audio processing too much. Audio is processed in fixed-size buffers. Here, we present a way of practically dealing with variable-size buffers.

3.1 General structure

In Antescofo, audio processing was previously handled in Puredata or Max/MSP, the host environment, or more rarely in synthesis environments such as SuperCollider, giving control of scheduling to these environment, without taking advantage of the additional information given by an augmented score. In new versions, audio processing tasks are beginning to be embedded into Antescofo [15], with Csound [43], Fluidsynth[11], or Faust [30]. Using Faust, which can compile to optimized code, makes it possible to target small single-board cards such as Raspberry Pi or Udoo.

DSP network Audio signals flow through an audio processing graph which connects audio processing nodes together. Audio links, which are independent from the audio processing tasks, can adapt the data flow to the specific needs of the nodes they link. Connections between audio can be dynamically changed in response of an event detected by Antescofo using the *patch* actions, which differ from PureData or Max/MSP where connections cannot be easily[12] changed during a performance. The audio processing nodes can be controlled using the control variables of Antescofo, so that parameters of an effect can change according to the tempo of the score, for instance.

Audio nodes can be *active* or *inactive*. If they are *inactive*, they are equivalent to an identity node. However, all processing tasks are started at the beginning of the score. There are four different sorts of audio links (see Fig. 7):

input to connect to audio inputs provided by the host environment

output to connect to audio outputs provided by the host environment

[11] `http://www.fluidsynth.org/`

[12] The usual workaround is to use a big matrix the lines and columns of which are the effects.

internal link to connect two audio processing effects

continuous variable to connect a curve in Antescofo to the audio buffers

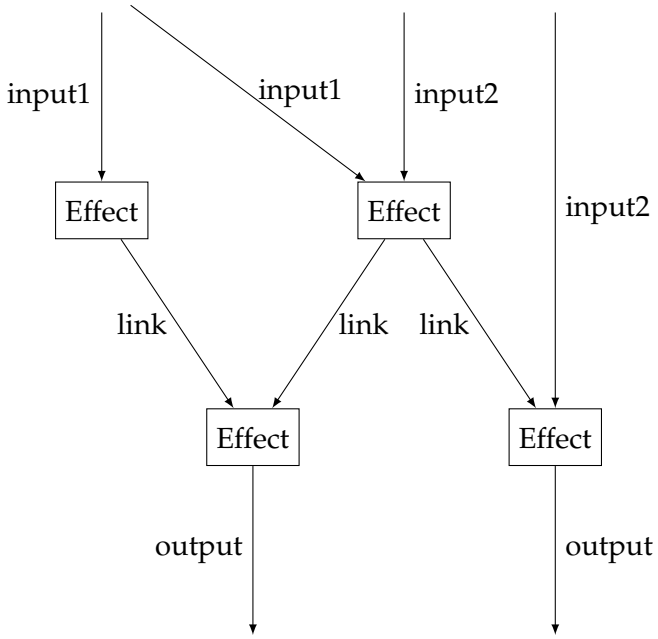


Figure 7: A DSP network.

The input and output of audio, through a periodic audio callback,[13] implies specific constraints on how to manage the rates of all the nodes with respect to the rate of the input/output nodes.

In the graph, data does not flow sample by sample, but buffer by buffer, and control happens at boundaries of the processing of a buffer, the usual separation between control rate and audio rate.

Continuous variables *Continuous variables* connect Antescofo discrete variables to the continuous signal[14]. A *continuous variable* can be assigned a constant value, or a curve. In practice, the sampling rate of the audio signal becomes the sampling rate of the curve.

Topological sort To determine the order of the audio processing, a topological sort is performed on the graph of effects. The graph of effect is assumed to be acyclic (otherwise, an error is thrown).

[13] Provided by the audio library, for example, CoreAudio on Mac OS X, Jack on Linux, or WASAPI on Windows.

[14] Actually, the signal is sampled, but at a much higher samplerate than that of curves in Antescofo.

Faust integration Faust effects are integrated into the Antescofo augmented score and compiled on the fly using a llvm embedded compiler [26]. Control happens at boundaries of Faust buffers, except in the case of continuous variables.

Listing 1 of *Anthèmes 2* involves Faust effects, such as:

```
1 $$linkHarm := faust::Harms($$audioIn,$h1,$h2,$h3,$h4,$hrout)
```

It corresponds to applying an harmonizer effect to the audio signal `audioIn` with control parameters given by `h1`, `h2`, ... `h4`, `hrout` and route the result to `linkHarm`.

4 Accommodating various rates

Our objective is to enhance this model by allowing more interleaving between control and audio.

4.1 Related work

A *dataflow* model of computation is a directed graph the nodes of which communicates by the channels represented by the edges of the graph. Each channel is a FIFO (First-in first-out) queue of tokens. The input and output rates of the nodes, *i.e.* the number of tokens they respectively consume and produce, are exposed as integers. The computations on a node are activated only when the required number of input tokens is attained.

The *dataflow* model is very effective to represent digital signal processing, and in fct environment such as Max or Puredata (see section 2.1) mimic the dataflow graph in their user interface.

They also lead to efficient implementations both for sequential and concurrent scheduling, especially when they can be statically scheduled.

Synchronous dataflow (SDF) [25] SDF is a *dataflow* model where the input and output rate is known before execution. Hence, this kind of *dataflow* is statically schedulable. Calculability is decidable and bounds on buffers for the channels are computable. The schedule is periodic.

Constraints on rates between nodes lead to linear equations that can be solved to find the periodic schedule if it exists.

Time-triggered synchronous dataflow (TTSDF) [1] Arumi claims that synchronous dataflow only addresses sequentiality and causality between consumed and produced data but does not take time into account, that's to say that there are events that triggers some audio processing, and that effects have to produce and consume before deadlines. Hence, he adapts synchronous dataflow to a scenario where input and output audio buffers are filled via a periodic callback. TTSDF ensures a fixed and optimal latency and remains statically schedulable, with a decidable computability. It is equally powerful as SDF.[15] It distinguishes *timed nodes* (input and outputs) and *untimed nodes* (the effects). TTSDF adds to the schedulability condition of SDF the condition each input and output nodes has to execute the same number of times per cycle.

[15] It means that all SDF graphs can be run by TTSDF graphs.

4.2 Buffer flow

Here, we also design a model inspired by SDF and TTSDF, but we try to better accommodate audio and control: we represent buffers, and not samples, as it is the case for a real implementation, and we define a property of concatenation that makes it possible to cut buffers in parts to interleave control when necessary.

4.2.1 Characterizing buffers

Definition 1 (Buffer)**.** *Let $\mathcal{E}$ an alphabet.*

A buffer $(b_i)_{i\in[\![1,n]\!]}$ *is a finite word of $n \geq 0$ elements of $\mathcal{E}$ such that $b = (e_i)_{i\in[\![1,n]\!]}$.*

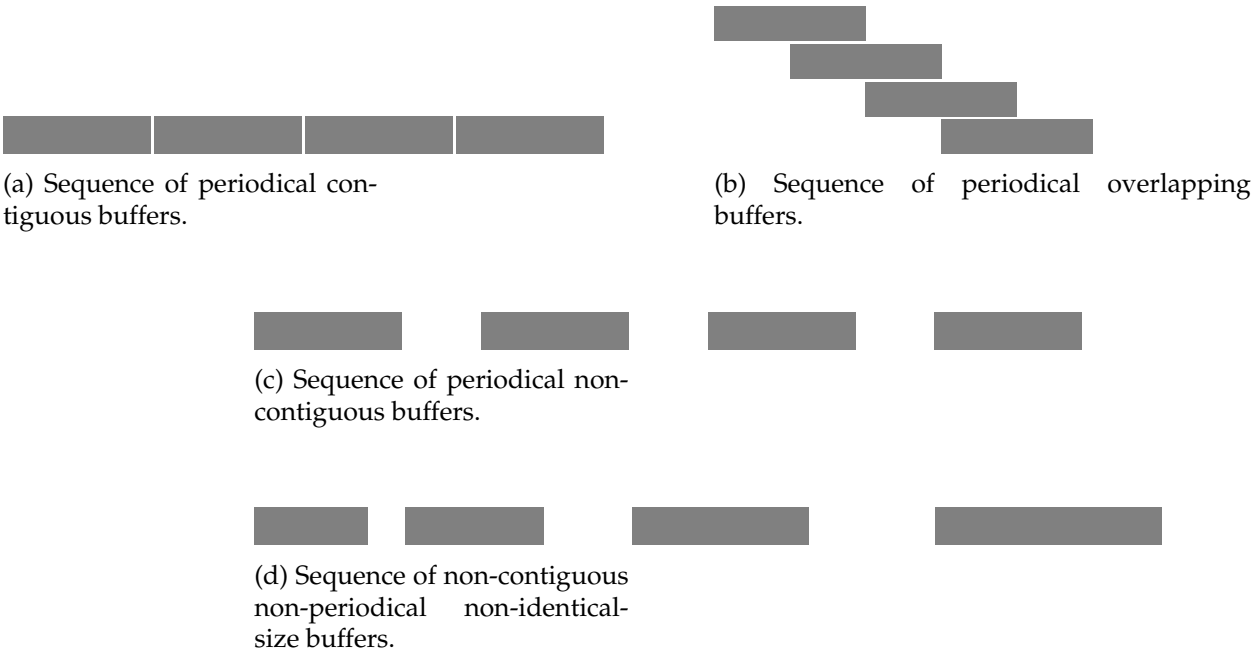
We will note $\mathbb{B}$ the set of buffers.

Definition 2 (Buffer type)**.** *We define the type of buffer $(e_i)_{i\in[\![1,n]\!]}$ where $(e_i)_{i\in[\![1,n]\!]} \in \mathcal{E}^n$ as $\mathcal{E}(n)$. We also consider the dependent type $mathcal E(x)$ where x is an integer variable.*

The buffer $(e_i)_{i\in[\![1,n]\!]}$ of type $\mathcal{E}_n$) will be noted:

$$(e_i)_{i\in[\![1,n]\!]} : \mathcal{E}(n)$$

Example. *Consider an audio graph which operates on a signal with elements of $\mathcal{A}$ in the time domain, and passes the signal through effects using a buffer b with a size of $\mathcal{A}64$ samples. The type of such a buffer is $\mathcal{A}_{64}$. We can also deduce the duration of processing a buffer if we know the samplerate ω: $d(b) = \frac{l(b)}{\omega}$.*



(a) Sequence of periodical contiguous buffers.

(b) Sequence of periodical overlapping buffers.

(c) Sequence of periodical non-contiguous buffers.

(d) Sequence of non-contiguous non-periodical non-identical-size buffers.

Figure 8: Various sequences of buffers.

Definition 3 (Type of a sequence of buffers)**.** *A sequence of buffers* $(b_i)_{i\in\mathbb{N}} \in \mathbb{B}^{\mathbb{N}}$ *is given the type* $(t_i)_{i\in\mathbb{N}}$ *where* t_i *are buffer types.*

Sequences of buffers represent the sequences of buffers that flow through effect nodes.

Example. *A sequence of buffer types.*

$$b(32)b(32)b(16)b(32)b(32)b(16)b(8)b(8)b(16)b(32)$$

Definition 4 (Periodic sequence of buffers.)**.** *A sequence of buffers* $(b_i)_{i\in\mathbb{N}} \in \mathbb{B}^{\mathbb{N}}$ *with type* $(t_i)_{i\in\mathbb{N}}$ *is* periodic *if and only if* (t_i) *is a periodic sequence.*

4.2.2 Concatenation of buffers

The intuition is that a sequence of buffers is a buffer, and a buffer is a sequence of buffers, so we can restructure a sequence of buffers into an equivalent sequence.

More formally, given a sequence of buffers (b_i) with the same element type $\mathcal{E}$. We have, for $j < j'$, $b = b_j \dots b'_j$ and b is a buffer. Reciprocally, if we have a buffer $b : \mathcal{E}(n)$, given $k \in [\![0, n]\!]$, it exists $b : \mathcal{E}(k)$ and $b' : \mathcal{E}(n-k)$ such that $b \cdot b'$ is a finite sequence of buffers.

4.2.3 Type coercion

Type coercion functions make it possible to resample, to convert from time domain to frequency domain and so on. We would like for instance from a sequence of temporal buffers, with audio samples, to get a sequence of spectral buffers, with bins.

Definition 5 (Conversion function)**.** *A sequence of buffers is transformed into another sequence of buffers of another type and another size.*

$$\begin{array}{rrcl} f: & \mathbb{B}^{\mathbb{N}} & \longrightarrow & \mathbb{B}^{\mathbb{N}} \\ & (b_i) : (\mathcal{E}(n_i)) & \longmapsto & (b'_j) : (\mathcal{E}'(n'_j)) \end{array}$$

Definition 6 (Buffer conversion function)**.** *A buffer can be transformed into several sequential buffers of various types, several buffers into one.*

$$\begin{array}{rrcl} g: & \mathbb{B}^{k} & \longrightarrow & \mathbb{B}^{k'} \\ & (b_i) : (\mathcal{E}(n_i)) & \longmapsto & (b'_j) : (\mathcal{E})(n'_j) \end{array}$$

Example. *A simple but useful buffer conversion function is:*

$$\begin{array}{rrcl} g: & \mathbb{B} & \longrightarrow & \mathbb{B} \\ & (e_i) : (\mathcal{E}, n) & \longmapsto & (e'_i) : (\mathcal{E}', n) \end{array}$$

It makes it possible to use a control as a signal, for instance.

Another useful transformation is resampling:

$$\begin{array}{rrcl} g: & \mathbb{B} & \longrightarrow & \mathbb{B} \\ & (e_i) : \mathcal{E}(n) & \longmapsto & (e'_i) : \mathcal{E}(n') \end{array}$$

The actual g can be done when oversampling by adding zeros between the original samples, and then low-filtering.

Another useful transformation is a conversion function between time domain and spectral domain (using the short-time Fourier transform, for instance).

The aim of these conversion functions is to declare types only for effects, and to automatically infer the right conversion functions between effects.

4.2.4 Buffer flow graph

Definition 7 (Buffer flow graph)**.** *A* buffer flow graph *is a directed acyclic graph $G = (V, E, \gamma)$. Nodes have precise input buffers and output buffers and edges are conversion functions,* i.e. *$V = \mathbb{B}^{\mathbb{B}}$. $E \in (\mathbb{B}^{\mathbb{B}})^2$ is the set of edges, and $\gamma : E \to \mathbb{B}^{\mathbb{B}}$.*

Nodes without incoming edges (called inlets*) are called* inputs *and nodes without outcoming edges (called* outlets*) are called* outputs*. Nodes with incoming and outcoming edges are called* effects*.*

Definition 8 (Types of nodes)**.** *A node with n inlets of types $t_1 \dots t_n$ and m outlets of types $t'_1 \dots t'_m$ has type:*

$$t_1 \times \dots \times t_n \to t'_1 \times \dots \times t'_n$$

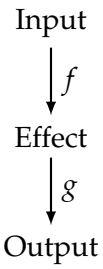


Figure 9: A buffer flow graph with one input node, one output node, and one effect node ; f and g are conversion functions.

Proposition (Well-typed buffer flow graph)**.** *A buffer flow graph $G = (V, E, \gamma)$ is* well-typed *if and only if for every edge $e = (n_1 : t_1, n_2 : t_2) \in E$, $\gamma(e)(n_1) : t2$.*

Merging edges

Definition 9 (Compatible edges)**.** *Let's consider two edges e_1 and e_2 that connect two nodes from the same node with conversion functions $f_1 : \mathcal{E}(k) \to \mathcal{E}(p)$ and $f_2 : \mathcal{E}_k \to \mathcal{E}'(p')$.*

e_1 and e_2 are compatible if $\mathcal{E} = \mathcal{E}'$.

If two edges e_1 and e_2 are *compatible*, we can rewrite the buffer flow graph as in Fig. 10, by adding an adaptor node, and using the following conversion functions (using the notation of definition 9, and notating $\mathcal{E}' = \mathcal{E}^1 = \mathcal{E}^2$): $g : \mathcal{E}_k \to {\mathcal{E}'}_{p \times p'}$, $f'_1 : {\mathcal{E}'}_{p \times p'} \to {\mathcal{E}'}_p$ and $f'_1 : {\mathcal{E}'}_{p \times p'} \to {\mathcal{E}'}_{p'}$.

This transformations is useful when the conversion from $\mathcal{E}$ to $\mathcal{E}'$ takes more time than just the transfer of the signal between nodes, that's to say, more time that the identity conversion function.

This transformation highlights that conversion functions can also be represented as nodes. We have chosen to represent them as edges to reflect the way the graph of effects will be defined by the user, by specifying effects, and let the conversion functions be inferred.

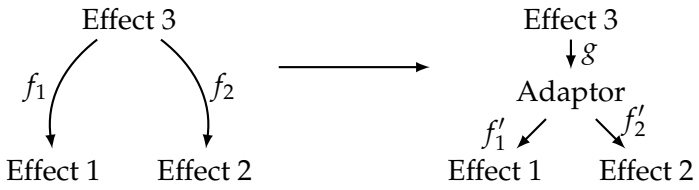


Figure 10: If f and f' are compatible, the two edges can be merged.

Type inference We want to only declare the types of effects, and as few as possible, and to infer the types of the other effects, as well as the conversion functions to apply between the effects. For that, we traverse the graph and collect constraints between each connected nodes. We insert a conversion function when two connected nodes input and output do not have the same type and these connected nodes have a type.

Some nodes, for instance, Faust effects, can work with any buffer size, so in this case, we constraint the conversion functions to work on the types of elements required by the node but sizes of buffers remain undetermined: such nodes are given types using a dependent type such as $\mathcal{E}(x)$ where x is a fresh variable. A solver for these constraints has started to be implemented.

Scheduling uses the same principles as TTSDF (see section 4.1).

Scheduling The static scheduling for audio with multiple rates has been started to be implemented, and there is now a version of Antescofo that should work soon on Udoo, a small electronic board.

These *buffer types* have two advantages:

- To adapt in the same audio graph effects that do not have the same input and output rates, which is not possible to do only with Faust, for instance.
- It is a first step to handle a control signal that can occur at arbitrary places during buffer processing.

Part III
Model-based analysis

The real-time emphasis of Antescofo [9], the way it reacts to input from the physical environment, makes Antescofo similar to embedded systems. We study here how to apply and adapt to IMS such as Antescofo some techniques and formalisms for testing and analysing embedded systems.

The models and algorithms described in this section have been designed with the case of Antescofo in mind (and in particular mixed music scores such as the one of the example of *Anthèmes 2* in Listing 1), but they are thought to be applicable to other IMS. In particular, we aim at improving the scheduling mechanisms and propose some analysis procedures for the system of the Patcher families, presented in Section 2.1, in collaboration with the developers of these systems (Cycling'74 for MAX and Miller Puckette for PureData).

5 Paradigms for real-time systems

In [23], the authors identify three main real-time programming paradigms:

Bounded execution time also called *scheduled model*, which is the classical model used in mainstream languages where execution times of programs have explicit deadlines.

Zero execution time also called *synchronous model*, used for synchronous reactive languages, such as Lustre [17], ReactiveML [29] or Esterel [3] (see Section 5.1). The execution time of a program including input, computations, and output is assumed to be zero.

Logical execution time also called *timed model* or *time-triggered* model, used in Giotto [18], or in xGiotto [14], which is event and time-triggered, (See Section 5.2). Input and output are assumed to be performed in zero time, but the processing time in between has a strictly positive fixed duration.

5.1 Synchronous languages

In ZET, execution time including input and output is assumed to be zero, or equivalently, the execution environment of the program is supposed to be infinitely fast. The associated paradigm is called *synchronous reactive programming*: *reactive* because it reacts with some output to some input (and it is an ongoing dialogue with the environment, unlikely to *transformational systems*) and *synchronous* because the reaction is executed in zero time (*soft-time*, software time, by opposition to *real-time*, the physical time, is always zero).

In a correct execution, a synchronous reactive program always writes an output before a new input is available : it has to be *deterministic*, that's to says it computes at most one reaction for any event and control state, and *reactive*, that's to say it computes at least one reaction for any event and control state (i.e. it terminates).

Hence, correctness analysis needs to check for the existence of fixed points, i.e. the absence of infinite cycles, using causality analysis. Besides code generation, compilers need to prove reactivity and synchrony.

ZET requires a thorough analysis of WCET – and in certain cases, the zero execution time hypothesis may be unrealistic – as well as proving the absence of infinite cycles.

Reactive synchronous programs usually describe a totally-ordered sequence of logical instants, contrary to Antescofo programs, where logical instants are partially ordered, because of multiple clocks that depend on a tempo value detected at runtime are coupled by a performance *tempo*.

Esterel [3]. Esterel is an imperative reactive synchrone language, and can express preemption and parallelism.

In the following code, `await` waits for two signals, and go to the next instruction if signal A or signal B is received.

```
[await A || await B];
emit O
```

The `emit` instruction is used to send signals.

Esterel has directly inspired the Antescofo language.

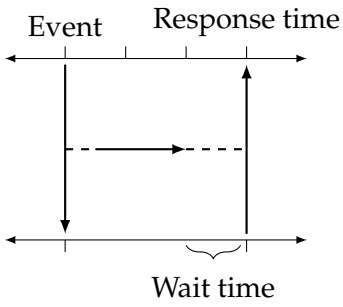


Figure 11: Logical execution time: delay if actual execution finishes before the pre-fixed execution time. Adapted from [22].

n-synchrone [31]. The *N-synchrone* paradigm is an extension of Lustre [17] that makes it possible to use clocks that are not necessarily synchronized, but *close* from each other, by using buffers, the size of which is automatically computed. For that, it uses a sub-typing constraint which is added to the clock calculus of Lustre. This approach works on periodic clocks, but also on clocks where only the mean rate and bounds on lateness or advances are known, by bounding these clocks by regular rates. This sort of approach can be related to finding bounds on tempo for an Antescofo score (see section 5.1).

5.2 Giotto and xGiotto

In LET [23], a program is assumed to read its input in zero time and write its output in zero time, but has a strictly positive fixed execution time. This duration is not an upper bound as a deadline in BET, it is the actual duration of the execution: in this model, soft-time is equal to real-time. A program is time-safe if there is enough time to compute the desired output, *i.e.* an output is not read by another task before it is written by the task in charge. The time safety can be checked at compile time using WCET evaluated for each task. An exception can be thrown at run time in case of error in analysis (e.g. wrong WCET). If a task program finishes early, it delays its output until the specified real-time for exec has elapsed (see Fig. 11). LET aims at separating environment time (interactions between software and physical processes) from platform time (the actual execution on a precise hardware, operating system) for the programmer point of view: it enables the programmer to focus only on environment time relieving them from the burden related to platform time (hardware performance, communication times, scheduling mechanisms and so on. The latter issues are delegated to a compiler and static analyser, in charge of proving formally that the code will execute correctly on the target platform.

Thanks to this time semantics, the behaviour of a time-safe program is only determined by the physical context. Using a faster machine only decreases machine utilization, thus increases concurrency (and does not lead to faster program execution). Hence this paradigm favours time determinism over performance.

5.2.1 Giotto: a time-triggered language

Giotto [23] is an actual language that uses the LET paradigm. In Giotto, codes are separated into functional code (tasks, written in C) with their specified execution time, system code (drivers)

which deals with input (sensors) and output (actuators) through ports, with zero execution time, and timing related instructions (E code) with zero execution time.

Giotto programs are built using one or several control modes in which tasks are periodically executed. Switching between modes is time-triggered, and Giotto programs can be in only one mode at a time. Then, they are compiled into E Code interpreted by a virtual machine (the E machine). The tasks in C are compiled [19] separately, and are dynamically linked by the E machine. E code generated from Giotto programs can be easily checked for schedulability: time safety of individual modes implies time safety of the whole program. Time-safe Giotto programs are time-deterministic.

As BET, Giotto programs need real-time scheduling, but do not require fixed-point analysis as ZET, and LET can be considered as an intermediate approach.

In the following code, we show a simplified Giotto program with 2 modes to control an helicopter:

```
start m_a {
mode m_a () period 20 {
actfreq 1 do p_a(d_a);
exitfreq 2 do m_b(c_b);
taskfreq 1 do t1(d_i);
taskfreq 2 do t2(d_s);
}
```

```
mode m_b() period 20 {
actfreq 1 do p_a(d_a);
exitfreq 2 do m_a(c_a);
taskfreq 1 do t_1(d_i);
taskfreq 2 do t'_2(d_s);
}
}
```

There are two tasks, given in native C code, the control task `t_1` and the navigation task `t_2`. `d_s` is a sensor driver which provides GPS data for `t_2`, `d_i` is a connection driver to send results of the navigation task `t_2` to the control task `t_1`, and `d_a` is an actuator driver for the actuator `p_a`. The instruction `actfreq 1` causes the actuator `p_a` to be updated once every 20 ms, and `taskfreq 2` implies that the navigation task is executed twice every 20ms.

Here, we have two modes: mode `m_a` represents the helicopter in hover mode, and `m_b`, in descent mode. The program starts in mode `m_a`. The conditions `c_a` and `c_b` are exit conditions from the modes and are evaluated twice every 20 ms (see `exitfreq 2 do m_b(c_b)`).

E code instructions Giotto programs are compiled into E code, which has the following instructions:

`call(d)` executes driver `d`

`release(t)` task `t` is given to the OS scheduler

`future(g,a)` E code at address `a` is scheduled for execution at time triggered by `g` (the period)

`jump(a)` unconditional jump to address `a`

`if(c,a)` conditional jump to address `a` if condition `c` verified

`return` terminates execution of E code

For the helicopter example, the Giotto program is compiled into the following E code:

```
a_1: call(d_a)
if(c_b, a'_3)
a'_1: call(d_s)
call(d_i)
release(t_1)
release(t_2)
future(g, a_2)
a_2: if(c_b, a'_4)
a'_2: call(ds)
release(t_2)
future(g, a_1)
```

```
a_3: call(d_a)
if(c_a,a'_1)
a'_3: call(d_s)
call(d_i)
release(t_1)
release(t'_2)
future(g,a_4)
a_4:if(c_a,a'_2)
a'_4: call(d_s)
release(t'_2)
future(g, a_3)
```

Code from `a_1` to `a'_2` implements `m_a` and code from `a_3` to `a'_4`, implements `m_b`.

An E code interpreter (E machine) maintains a program counter, the status of the tasks, and a queue of trigger bindings (trigger, address), as added by the `future` instruction. For scheduling, it uses the platform scheduler or can alternatively compile scheduling decisions into *schedule carrying code* [21] as a specific set of instructions (S code).

5.2.2 xGiotto: a time and event-triggered language

xGiotto [14] adds event-triggering to the time-triggered approach of Giotto: xGiotto can support asynchronous events and uses event scoping, handling events within a hierarchical block structure. An event can be a clock tick or a sensor interrupt.

xGiotto uses the following instructions:

`react {b} until [e]` : called reaction block; `b` is the body (sequence of triggers, releases, reactions); `e` is an event for termination.

`release t (in) (out)` : `t` is a task released to the scheduler; `in`, a list of input ports, `out`, a list of output ports for the results of the task.

`when [e] r and whenever [e] r` : `e` is an event indicating when to start invoking reaction `r` (or a composition of reactions); `e` can be propagated to passive scopes: with attribute *forget*, it is ignored, with *remember*, it is postponed until its scope becomes active again, or it may disable descendant blocks in passive scope with *asap*, using the syntax `attribute e`. With `whenever`, the statement is re-enabled immediately after the event occurs.

An event scope is composed of the `until` event of the reaction block and of the `when` events of the when statements inside the reaction block. Once reactions of `when` statements are executed, the current event scope is pushed onto a stack and becomes *passive*; new event scopes are created, which becomes *active* scopes, thus leading to a tree of scopes, the leaves of which are the active scope, and the root, the initial scope. Parallel composition of reaction blocks can be wait-parallel (||) or asap-parallel (&&). For wait-parallelism, reaction blocks finish once all events have occurred, whereas for asap-parallelism, the `until` event of any of the reaction blocks disables the sibling blocks.

The *trigger queue* stores the enabled reactions with their triggering event ; tasks that have been released in a scope are contained in the *ready set*.

In the following example, where we omit input and ouput ports for simplicity, the xGiotto code on the right has the same behaviour as the Giotto program on the left. In the xGiotto code, task `t1` is released initially and the inner block `react ... until` which releases task `t2` is executed. The event that repeats every 10 ms is notated `10`.

```
mode m() period 20 {
    taskfreq 1 do t1();
    taskfreq2 do t2();
}
```

```
react {
    release t1() ();
    begin
        react {
            release t2()();
        } until [10];
        react {
            release t2 () ();
        } until [10];
    end
} until [20];
```

xGiotto programs can also be compiled to S code and E code: asynchronous events can be handled by triggers in E code.

Program analysis. The xGiotto compiler performs several program static analyses.

Race detection. A *race* occurs when two ports are updated at the same time. A race-free program will be value-deterministic. Races could be detected by a traversal of the configuration graph, which is exponential. The compiler actually performs a conservative polynomial-time check: a set of potential termination events is associated to every reaction block. If for two distinct release statements with a common output port, the potential termination events associated are disjoint, then, all program traces are race-free.

Resource requirements. The memory of embedded systems is often constrained so the compiler estimates conservative bounds on the trigger queues, and the size of the scope tree and ready set.

Time safety. It uses a two-player safety game. We adapt this time-safety game to our intermediate language in section 9.

BET and ZET are more general paradigms than LET, but restraining to LET makes it possible to have a model that is value and time-deterministic. These properties are fundamental for music embedded systems. In table 2, we show how languages using the Zero Execution Time paradigm, and languages using the Logical Execution Time paradigm, compare.

6 Intermediate representation

Our intermediate representation is an adaptation of Giotto and xGiotto, and in particular E code (see [20]), to multiple coupled clocks, and of an intermediate representation [33] of Antescofo

	Temporal events	Arbitrary events	ZET or LET	Several clocks
Esterel	Yes	Yes	ZET	Yes
n-synchrone	Yes	Yes	ZET	Yes
Giotto	Yes	No	LET	No
xGiotto	Yes	Yes	LET	No
Antescofo	Yes	Yes	Rather LET	Yes

Table 2: Comparisons of some real-time programming languages. A *temporal event* refers to waiting for a precise duration.

inspired by timed automata models (see [11]) to specify formally the reactive engine of Antescofo to perform model-based testing, that does not take tasks into account. We will use the extended intermediate representation to perform a static analysis to test the schedulability of an Antescofo program. For the sake of readability, we use below a graph representation of the intermediate representation.

6.1 Syntax

Program. An IR program extended for task management is a septuplet $(P, T, D, Addr, a_0, ins, next)$ where:

P a set of global variables, which can be environment variables, P_E, and task variables, P_T.

T a set of tasks. A *task* t is a piece of C code (with restrictions, see below) linked to the IR by a quadruplet $(P[t], I[t], f[t], w)$ such that $P[t] \subset P_T$, $I[t] \subset P_E$, $f[t] : I[t] \cup P[t] \mapsto P[t]$ and $w \in \mathbb{R}$ the worst case execution time of the task.

$Addr$ a set of addresses

a_0 is the initial address

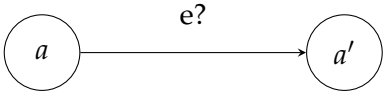
$ins(a, a')$ is the instruction on labelling the edge $a \to a'$.

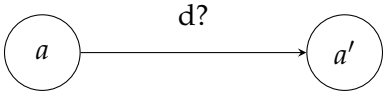
$next(a) : Addr \to \mathcal{P}(Addr)$ is the set of addresses outgoing from $Addr$. The addresses which are connected to address a in the graph of instructions.

The global variables of P play the role of the ports in Giotto.

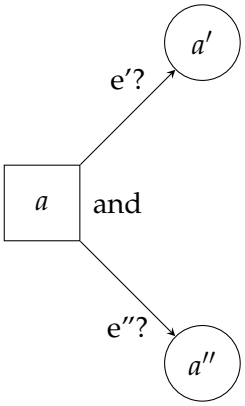
Instructions:

Wait on an event. Go to $next(a)$ on activation of a trigger. e is a boolean expression on the environment variables. d represents the particular case of a *time event*, when a clock variable has elapsed for d time units (see Section 6.2).

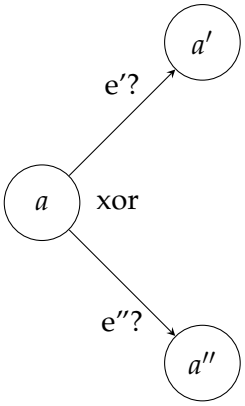




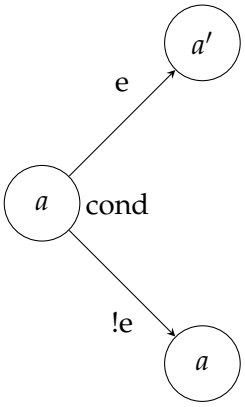
and. Parallel execution. The control is duplicated from address a to both a' and a'' which are reached the respective conditions e' and e'' realize. When one event (amongst e' and e'') occurs, take the corresponding branch and continue waiting for the other.

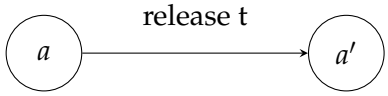


xor. Wait concurrently for events e' and e'', and cancel waiting when one of them happens. As soon as one of the event occurs, take the corresponding branch and stop waiting for the other.

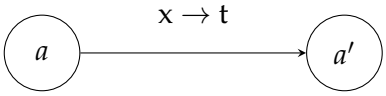


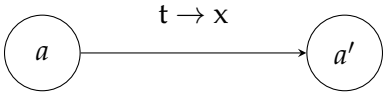
Condition e is a boolean condition.



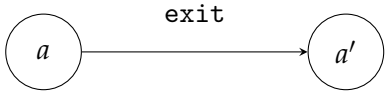
Release a task. 

Read or write a variable. $x \to t$ is the copy, by task t, of the global variable x (either environment variable or task variable) into one of its local variables, and $t \to x$ is the inverse operation.





These instruction can be used by tasks to communicate with each other and with the environment. It must be done with the intermediate of global variables: tasks cannot write directly on the local variables of another task *i.e.* We cannot write $t' \to t$.

Exit the program. 

The three first instructions `wait`, `and`, `xor` are called *asynchronous* (or blocking): the time flows during their execution. The other instructions are called *synchronous* (or instant): they are assumed to execute in zero time.

Remember that tasks are written separately in C code. In this model, an important characteristic of tasks is that they are isolated spatially and temporally. In their C code, tasks are not allowed to include instructions to synchronize to each other or wait for time. In particular, they must not include system calls. For synchronization, tasks must use the above instructions $x \to t$ and $t \to x$ and for waiting, the `wait` instruction.

Fig. 12 presents the intermediate representation associated to the Antescofo program of listing 1.

6.2 Variables and clocks

Among the environment variables in P_E, there are *clocks* which can represent physical time or musical time (tempo). We will note p_{c} the clock representing physical time and p_{m} the musical clocks. There is also a variable representing musical events (notes, chords, trills and so on) which will be noted q_{n}.

We suppose the following integral relation between physical time and musical time:

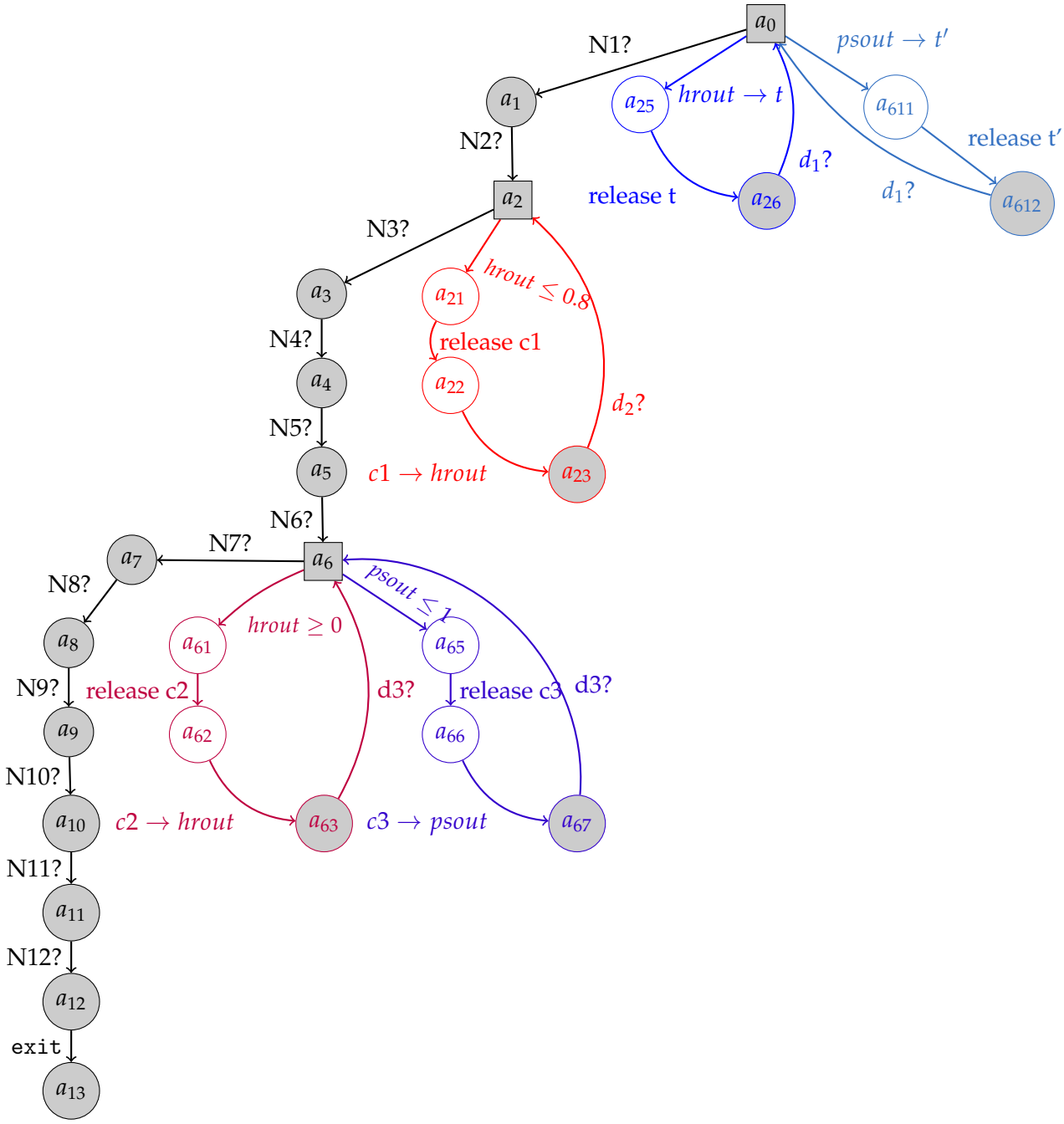
$$p_{\mathsf{c}} = \int_0^{p_{\mathsf{m}}} \frac{1}{\omega(m)}\, dm \tag{1}$$

where ω is the instantaneous tempo. A first approximation is to take ω piecewise constant (see Eq. 2). For instance, if tempo is ♩= 60, the duration associated to a whole note is $\frac{4}{60} \times 60 = 4$s.

$$p_{\mathsf{c}} = \frac{p_{\mathsf{m}}}{\omega} \tag{2}$$

Other conversion relations are presented in table 3.

An *input event* is a change of an environment port or a task port.



N1: NOTE 0 1.0
N2: NOTE 8100 0.1
N3: NOTE 7300 1.0
N4: NOTE 7100 0.1
N5: NOTE 7200 1.0
N6: NOTE 6600 1/7

N7: NOTE 6900 1/7
N8: NOTE 7200 1/7
N9: NOTE 7300 1/7
N10: NOTE 7400 1/7
N11: NOTE 7500 1/7
N12: NOTE 7700 1/7

Figure 12: Intermediate code for the beginning of Anthemes II. Logical instants are drawn with a grey background.

	musical time $p_{\mathsf{m}}, \omega(t)$	physical time p_{c}
musical time $p'_{\mathsf{m}}\omega'(t)$	$p'_m = \int_0^{p_{\mathsf{m}}} \frac{\omega(b)}{\omega'(b)}\, db$	$p'_m = \int_0^{p_{\mathsf{c}}} \omega'(t)\, dt$
physical time p_{c}	$p_{\mathsf{c}} = \int_0^{p_{\mathsf{m}}} \frac{1}{\omega(b)}\, db$	$\times$

Table 3: Conversion functions between musical time and physical time. The table reads from top labels to left labels.

6.3 Semantics

A configuration is a septuplet $c = (r, n, s, a, A_{\mathsf{s}}, \mathcal{T}, \mathit{Tasks})$ where $r \in \mathbb{R}^+$ is the date of a *logical instant* in physical time, $n \in \mathbb{N}$ a step number used to order the events occurring in the same logical instant (see the *superdense* model of time [34]), $\sigma : P \to V$ is the environment of *global variables*, $a \in \mathit{Addr} \cup \{\odot\}$ is the current address ($\odot$ means that we do not execute an instruction for this transition.), $A_{\mathsf{s}} \subset \mathit{Addr}$ a set of addresses of synchronous instructions, $\mathcal{T}$ a set of meta-triggers (G, σ) with, G a set of triggers (g, a), where g is the boolean condition to test, and σ the environment of variables when a trigger is added, *Tasks* a set of pairs (t, σ') called *task set*, where t is a task, σ' a variable environment of P.

A configuration c is *initial* if $r = 0$, $n = 0$, $a = a_0$, $A_{\mathsf{s}} = \{\mathit{next}(a_0)\}$, $\mathcal{T} = \emptyset$ and $\mathit{Tasks} = \emptyset$.

By abuse, we will later use the name of a variable as its value if there is not ambiguity on the environment.

Updating variables. We note $\sigma[v \mapsto x]$ the update of variable x with the value v in the environment. We note $\mathit{vars}(t)$ the set of tasks variables related to task t. $s[v \mapsto \mathit{vars}(t)]$ is a notation for assigning value v to one non specified variables of t.

Updating clocks. All clocks have to be synchronized together, that's to say, if one clock is updated, the others have to be updated. Hence, when updating a clock, $s[t \mapsto p_{\mathsf{c}}]$, we suppose we update the other clocks according the the conversions of table 3.

Rules. The following rules define the transitions between configurations.

Environment transitions.

$$\langle r, n, \sigma, \odot, A_{\mathsf{s}}, \mathcal{T}, \mathit{Tasks}\rangle \longrightarrow \langle r', 0, \sigma[v \mapsto X], \odot, A_{\mathsf{s}}, \mathcal{T}, \mathit{Tasks}\rangle \textbf{ Update of a variable}$$

which is not a clock

$$\langle r, n, \sigma, \odot, A_{\mathsf{s}}, \mathcal{T}, \mathit{Tasks}\rangle \longrightarrow \langle r', 0, \sigma[r' \mapsto p_c], \odot, A_{\mathsf{s}}, \mathcal{T}, \mathit{Tasks}\rangle \text{ where } r' > r \textbf{ Time elapse}$$

$$\langle r, n, \sigma, \odot, A_{\mathsf{s}}, \mathcal{T}, \mathit{Tasks}\rangle \longrightarrow \langle r, 0, \sigma, \odot, A_{\mathsf{s}}, \mathcal{T}, \mathit{Tasks}'\rangle \text{ where } r' > r \textbf{ End of a task}$$

where $\mathit{Task}' = \mathit{Task} \setminus \{(t, \sigma')\}$

Event-triggered transitions.

$$\langle r, n, \sigma, a, a' :: A_s, \mathcal{T}, Tasks\rangle \longrightarrow \langle r, n+1, \sigma, a', A_s, \mathcal{T} \cup \{(\{x, a'\}, \sigma)\}, Tasks\rangle$$
if $ins(a, a') = x?$ **Adding trigger**

$\langle r, 0, \sigma, \odot, A_s, (G, \sigma') :: \mathcal{T}, Tasks\rangle \longrightarrow \langle r, 0, \sigma, a', next(a') \cup A_s, \mathcal{T}, Tasks\rangle$ if there exists a $(g, a') \in G$ such that g is true on (σ, σ') and $\sigma \neq \sigma'$ **Activation of a trigger**

We require that $\sigma \neq \sigma'$ so that a trigger does not execute on the same logical instant as the one when it is added to the trigger queue, to prevent causality problems from occurring.

For instance, for a time-trigger on the physical clock for a duration d, the trigger g is $\sigma(p_c) = \sigma'(p_c) + d$.

$$\langle r, n, \sigma, a, A_s, \mathcal{T}, Tasks\rangle \longrightarrow \langle r, n+1, \sigma, \odot, A_s, \mathcal{T} \cup_{a' \in next(a)} \{(x, a', \sigma)\}, Tasks\rangle$$
if $\forall a' \in next(a), ins(a, a') = x?$ and $ins(a) =$ xor **Adding xor trigger**

Execution of synchronous instructions.

$\langle r, n, \sigma, a, a' :: A_s, \mathcal{T}, Tasks\rangle \longrightarrow \langle r, n+1, \sigma, a', next(a) :: A_s, \mathcal{T}, \{(t, \sigma)\} \cup Tasks\rangle$ if $ins(a, a') =$ `release` t
Release of a task to the scheduler.

$$\langle r, n, \sigma, a, a' :: A_s, \mathcal{T}, Tasks\rangle \to \langle r, n+1, \sigma[X \mapsto vars(t)], a', next(a) \cup A_s, \mathcal{T}, Tasks\rangle$$
if $ins(a) = X \to t$ **Write of an environment variable into a task variable.**

$\langle r, n, \sigma, a, a' :: A_s, \mathcal{T}, Tasks\rangle \longrightarrow \langle r, n+1, \sigma, a', \{a''\} \cup A_s, \mathcal{T}, Tasks\rangle$ if $ins(a) = cond(b)$, $ins(a, a'') = b$ and b is $true$ **Condition, true**

$\langle r, n, \sigma, a, a' :: A_s, \mathcal{T}, Tasks\rangle \longrightarrow \langle r, n+1, \sigma, a', \{a''\} \cup A_s, \mathcal{T}, Tasks\rangle$ if $ins(a) = cond(b)$, $ins(a, a'') = !b,$ and b is $false$ **Condition, false**

Program termination. A program finishes if there are no more instructions to execute, or if we encounter the `exit` instruction.

$$\langle r, n, \sigma, a_0, \emptyset, \mathcal{T}, Tasks\rangle \xrightarrow{*} \langle r', n', \sigma', a, A_s, \mathcal{T}', Tasks'\rangle$$
with $ins(a) = exit$ or $A_s = \emptyset$ and $\mathcal{T} = \emptyset$

6.4 IR generation from Antescofo programs

Antescofo programs, as presented in Sect. 2.8, can be transformed into IR, with a separation into a set of computation tasks in C and IR instructions. This has been implemented for a fragment of Antescofo DSL in the case of the simpler IR of [33], which does not contain tasks. In our work, we have done the transformation manually on examples. The implementation of a complete transformation is further work. The generation of IR from programs of other IMS such as MAX or Pure Data, in order to benefit from the analyse procedures presented below, is also to be investigated.

Without unfolding, the size of the IR is linear[16] with the size of Antescofo code.

Generating IR code for audio processing tasks. Audio processing tasks have to read input audio buffers and write output audio buffers periodically. The period for buffers of identical sizes n and samplerates f is $d = \frac{n}{f}$. We note the periodic task t, the input buffer, $input$, and the output buffer, $output$. The generated IR is given on Fig. 13.

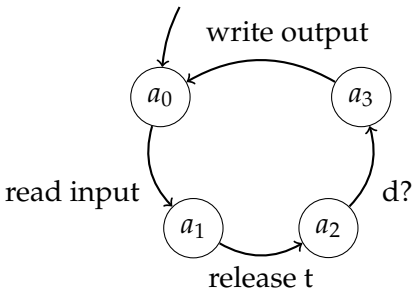


Figure 13: Intermediate code for an audio processing task.

Unfolding loops and curves. Curves and loops can be modelled using cycles and conditions, as shown on Fig. 14. However, when doing tests, we would not be able to test the condition on the variables because tasks are black boxes. Nevertheless, we can know on an Antescofo code how many times the body of the loop is executed. On listing 2, `Curve` performs a linear interpolation of variable `hrout` between 0 and 0.8 with a $0.1ms$ grain, that's to say 250 executions of the body.

```
1   Curve c1
2   @grain := 0.1
3   {; bring level up to 0.8 in 25
4   $hrout
5   {
6   {0}
7   25ms {0.8}
8   }
9   }
10
```

Listing 2: "Curve in Antescofo"

[16] See [32] for a proof for a similar version of the IR code without tasks.

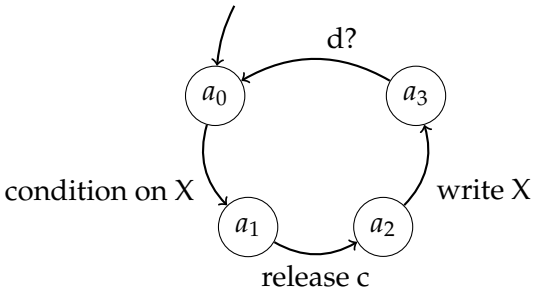


Figure 14: Intermediate code for a curve.

Instead of using the intermediate code of Fig. 14, we unfold this loop into a chain the size of which is the required number of executions of the loop body. However, for long scores, unfolding a loop can yield a huge number of nodes. For an audio processing task with a typical 1.45ms period, the order of magnitude of the unfolding for 1 minute is 10^5 nodes. Moreover, this simple unfolding is not possible for dynamic periods[17] so we assume that in the following sections, all periods are static.

7 Schedulability test

Here, we present an algorithm that performs a static analysis on the intermediate representation of an Antescofo program defined in the previous section, in order to test its schedulability given precise *tempi*.

Definition 10 (Logical instant)**.** *A* logical instant *is an instant in which all synchronisation instructions are executed instantaneously. A new logical instant starts for each asynchronous instructions.*

In other words, in run *(a sequence of configurations with transitions between them), a* logical instant *is the the first configuration in a* run *starting by a* `wait` *or a* `xor`*.*

Note that logical instants are partially ordered : $a \prec a'$ if and only if there exists a path between a and a', where a and a' are the current address in the configuration associated to the logical instants. We note $succ(a)$ the set of the smallest upper bounds of a for the order relation *path between logical instants*.

Definition 11 (Tempo function)**.** *A* tempo function $\omega : Addr \times Addr \to \{\spadesuit\} \cup (\mathbb{R} \to \mathbb{R})$ *is such that for a, a' two asynchronous instructions, $\omega(a, a')$ is a continuous function (which gives the instantaneous tempo) if $a' \in succ(a)$, otherwise, $\omega(a, a') = \spadesuit$ (undefined tempo).*

In other words, the tempo function *only changes on* logical instants*.*

In what follows, we assume given a predefined tempo function ω. It can be for instance the tempo function that corresponds to an ideal performance (*i.e.* a performance following strictly the tempo delays in the score). We also suppose that for every a, a' such that $\omega(a, a') \neq \spadesuit$, $\omega(a, a')$ is a constant function. We will note $\omega(a, a') = \text{cste}$ instead of $\omega(a, a') = t \mapsto \text{cste}$.

[17] In Antescofo, and in our IR, the waiting duration can be a variable.

A program is *time-safe* if there is enough time to compute the desired output, *i.e.* an output is not read by another task before it is written by the task in charge. We aim at verifying whether the set *Tasks* of active tasks is schedulable given their WCET and deadlines for execution and whether the program is *time-safe*. For that purpose, we need to determine what tasks are executing at a given logical instant.

As we know the tempo between every asynchronous actions, we can convert all durations in musical time into durations in physical time. We also need to know the deadline (in physical time) of the tasks. We have supposed that we know the worst case execution time of the tasks. With the deadlines and the WCET, we can apply a scheduling test to check the schedulability of the tasks.

Algorithm 1 Schedulability test.

Input: Antescofo program P, a tempo function τ
Tasks[a, t] : tasks being executed at the first address a of a logical instant where logical instants are ordered with respect to physical time t
s[ta] start time of task ta
d[ta] deadline of task ta. Initial value: $+\infty$
function TRAVERSAL(a, t)
  $a_l \leftarrow$ GETFIRSTADDRESS(a) ▷ Get first address associated to logical instant
  Tasks[a_l, t] is initialized with the tasks that were running just before
  for all $a' \in next(a)$ **do**
    if $ins(a, a') =$ `release ta` **then**
      Tasks[a_l, t] $\leftarrow$ Tasks[a_l, t] $\cup$ ta
      s[a] $\leftarrow t$
    else if $ins(a, a') = ta \rightarrow X$ **then**
      if s[ta] $<$ d[ta] **then**
        d[ta] $\leftarrow s$[ta]
      end if
    else if $ins(a, a') = d$? **then**
      $t \leftarrow t+$CONVERT(d, τ) ▷ Identity if d is in physical time.
    end if
    if $ins(a, a') \neq$ `exit` **then**
      TRAVERSAL(a', t)
    end if
  end for
  return
end function
TRAVERSAL(a_0, 0)
REMOVETASK(a_0) ▷ Remove occurence of task ta in Tasks[a, t] if $t >$ d[ta] for all ta in Tasks
for all a logical instant, in the order of increasing t **do**
  if not SCHEDULINGTEST(Tasks[a, t], WCETs) **then**
    stop and fail
  end if
  succeed
end for

Read and writes of variables can entail dependencies. In this case, we could use a scheduling test for tasks with dependencies, such as EDF*. Here, the scheduling test is indeed a black box. For audio processing that uses a dataflow graph, we can determine a static scheduling of the tasks.

Finding deadlines. We determine deadlines so that we ensure time-safety, that's to say, a ready task is not interrupted by input and output. That's why we can set the deadline of a task as the logical instant where the first write that it needs happens.

Hence, to determine this deadline, we traverse the graph from the logical instant which happens at the same time of logical instant of `release` t instruction until the first read or write related to t. We assume that every task has at least one associated read or write.[18]

For audio processing tasks (see section 6.4), we can thus deduce the deadline, which is the duration to process an audio buffer.

Complexity. The algorithm consists of a fixed number of graph traversals. Hence, the complexity of checking the time-safety at a given logical instant is linear in the size of the graph.

Correctness of the algorithm. An IR program for which the schedulability test succeeds is *time-safe*. We have found a total order based on physical time, instead on the previous partial order. The utilization tests performed at every logical instants, which have been ordered by physical time, is similar to a rejection test [7] for aperiodic hard-real time tasks, where tasks are rejected if acceptance leads to a *time-safety* violation.

Theorem 1 (Correctness of time-safety checking)**.** *If an IR program* Π *passes the schedulability test, then it is* time safe*.*

Proof. Let's consider a program with tasks $T_1, \dots T_p$ such that for every musical instant, the schedulability test succeeded. We are going to exhibit a feasible schedule for this Antescofo program and these tasks.

We note $a_1 \dots a_n$ the list of logical instants and $t_1, \dots t_n$ their associated physical time. We also note the release time and the deadlines of a task T respectively $T.r$ and $T.d$.

For every $k \in [\![1, n]\!]$, as the schedulability test as succeeded, we have a feasible schedule s_k where triples are (task, start, duration):

$$((T_{j_1^k}, t_k^1, d_k^1), \dots (T_{j_q^k}, t_k^q, d_k^q))$$

with $t_k^1 < \dots < t_k^q$ and $j_1^k, \dots j_q^k \in [\![1, p]\!]$

The scheduling algorithm updates the schedule at each logical instant. A *feasible schedule* for the whole execution is the concatenation of *feasible schedules* s_k. □

Schedulability test on the example. The specified tempo of *Anthèmes 2* (see 2.8 for the Antescofo code) is ♩= 46. Hence we can deduce the physical durations of the notes (see Fig. 4); d_2 lasts 0.13 ms.

The IR with physical durations instead of musical durations is presented on Fig. 15. w_t and $w_{t'}$ are respectively the worst case execution times of tasks t (dark blue on the graph) and t' (light blue on the graph).

[18] Which is a reasonable assumption: tasks usually interacts with the environment or other tasks; otherwise, they are useless.

	N1	N2	N3	N4	N5	N6	N7	N8	N9	N10	N11	N12
Musical time (beats)	1	0.1	1	0.1	1	1/7	1/7	1/7	1/7	1/7	1/7	1/7
Physical time (s)	1.3	0.13	1.3	0.13	1.3	0.18	0.18	0.18	0.18	0.18	0.18	0.18

Table 4: Physical duration of musical events for the beginning of Anthèmes 2 by Pierre Boulez if the tempo does not change from the specified value ♩= 46.

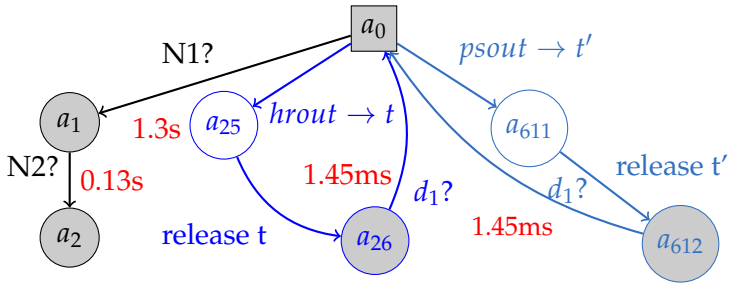


Figure 15: Beginning of Anthèmes 2 with physical durations.

On a_1, our algorithm detects that tasks t and t' are active, so we want to test, if we use EDF:

$$\frac{w_t}{d_1} + \frac{w_{t'}}{d_1} < 1$$

8 Bounds on tempo

Let us assume that tasks consume their whole WCET.

The algorithm proceeds as follows: from a logical instant a, it traverses back and add active tasks for this logical instant a as long as the utilization test succeeds on that node, and determines on each asynchronous edge which uses a musical time what the maximum tempo is. We can also deduce other bounds from the utilization test. At the same time, it traverses the graph for the first node, forward, because audio tasks are typically launched at the very beginning of an Antescofo score (see section 3) and an only backward approach would entail to traverse backward until this first node.

Example. For instance, let us find tempo bounds for the logical instant a_2 on Fig. 16: tasks t and t' are released to the scheduler every $d1$, and hence are active at a_2. Task $c1$ is released at $a2$.

We note d'_2 the physical duration associated to $d2$: $d'_2 = \frac{d_2}{\omega(a_{23},a_2)}$. w_1 is the worst case execution time of task c_1 (red on Fig. 16).

A first condition is $w_1 < d'_2$, hence $\omega(a_{23}, a_2) < \frac{d_2}{w1}$. The utilization test for EDF gives a tighter bound:

$$\frac{w_1}{d'_2} + \frac{w_t}{d_1} + \frac{w_{t'}}{d_1} < 1$$

from which we can deduce, given the relation between d_2 and d'_2:

$$\omega(a_{23}, a_2) < \omega_{\max} = \frac{d_2}{w_1}(1 - \frac{w_t + w_{t'}}{d_1})$$

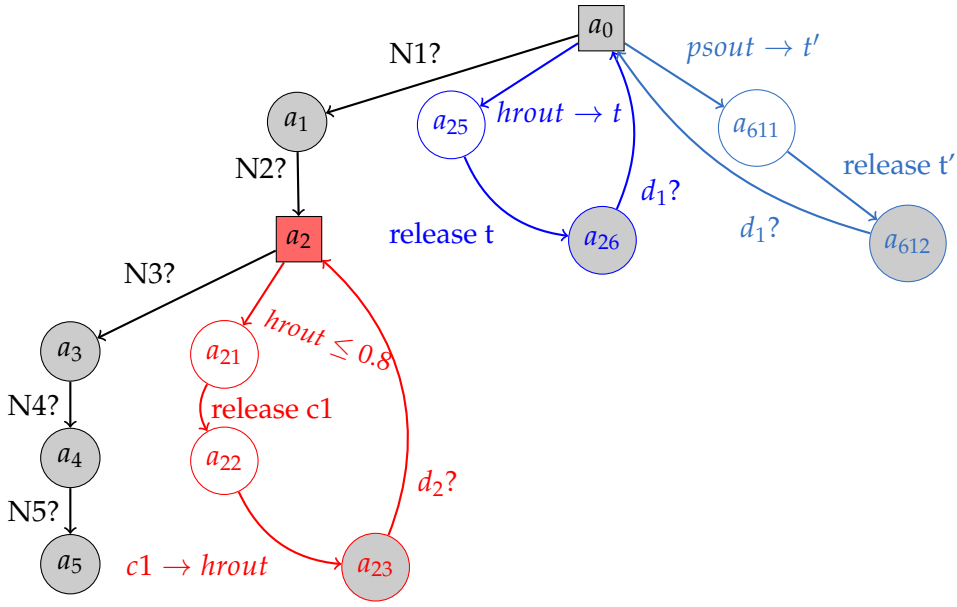


Figure 16: Bounds on tempo for node a_2.

We find other bounds similarly as for the first condition, and to find the global bound for the tempo, we take the minimum of these upper bounds.

Here, $d_1 = 1.45ms$ (for $44100Hz$ samplerate, and a buffer size of 6), $d2 = 0.1$ beat. We cannot easily know the WCET of the tasks, so let's assume that $w_1 = 0.001ms$[19]. Thus, the first bound is 100 beats per second, which is totally unrealistic. However, for more computationally-demanding tasks, for instance, with $w_1 = 0.1ms$, the upper bound becomes 1 beat per second, that's to say ♩= 60, which is a tempo commonly found in scores.

Existence and tightness of a bound. If there is no involved musical time, we can deduce nothing about the tempo; indeed, there is no tempo.

The upper bound on tempo depends a lot on the WCET of tasks, which is difficult to determine on mainstream operating systems and processors such as Mac OS X, Windows, and Linux, due to the caches and the non-deterministic input/output primitives of the operating system, so it is likely the upper-bound is not tight.

The tempo function which is calculated here is piece-wise constant, which means that the performance has brutal change of tempi. Discontinuous change of tempi are not uncommon in music, but it has also *accelerando* and *ralentendo*[20]. To get a smoother tempo, a polynomial interpolation could be applied between the extreme values found by our method. Some studies [16] claim that these changes of tempi are logarithmic.

[19]The task here calculates a linear interpolation, which is very quick, and likely to be negligible with respect to d_1.
[20]Where tempo accelerates or deccelerates.

9 Two-player safety game

We use a two-player safety game to draw some theoretical results on the schedulability of an Antescofo program compiled to the intermediate representation. The scheduling strategies we will get with the game could also be used to schedule an Antescofo program in practice. This two-player safety game is very similar to the one presented in [20]; the main difference is that several clocks have to be taken into account.

We modify the configurations $c = (r, n, \sigma, a, A_s, \mathcal{T}, \mathit{Tasks})$ in section 6.3 as follows: *Tasks* is now a set of triplets (r, σ', Δ) where $\Delta \in \mathbb{R}^+$ is the amount of CPU time the task has run.

We replace the rules of time elapse and of task removal of section 6.3 by the following rules:

Time elapse with idle CPU Change of a clock variable.

$$\langle r, n, \sigma, \odot, A_s, \mathcal{T}, \mathit{Tasks}\rangle \longrightarrow \langle r', 0, \sigma[r' \mapsto p_c], \odot, A_s, \mathcal{T}, \mathit{Tasks}\rangle \text{ where } r' > r$$

Time elapse with used CPU Change of a clock variable and execution of a task in the ready set. If the task has consumed all its WCET, it is removed from the ready task set.

$$\langle r, n, \sigma, \odot, A_s, \mathcal{T}, \mathit{Tasks}\rangle \longrightarrow \langle r', 0, \sigma[r' \mapsto p_c], \odot, A_s, \mathcal{T}, \mathit{Tasks}'\rangle \text{ where } r' > r$$

where $\mathit{Tasks}' = \cup_{(t,\sigma',\Delta)\in \mathit{Tasks}, \Delta \leq w_t} \{(t, \sigma', \Delta + r' - r)\}$

The other rules of section 6.3 are kept. When releasing a task, we set $\Delta = 0$.

A trace is a sequence of configurations such that there is an environment event, a time elapse with idle CPU, a time elapse with used CPU, or an IR instruction, between two configurations.

Two-player safety game A *two-player safety game* is played on an arena $\mathcal{A} = (V, E, c)$ where (V, E) is a graph, and $c : V \to C$ a colouring function. Vertices of V are partitioned into two sets V_1 and V_2 corresponding to two players.

Game A *play* π is an infinite word of consecutive vertices $v_0 v_1 \ldots$, *i.e.* for all $i \in \mathbb{N}$, $E(v_i, v_{i+1})$. A play π induces a sequence of colours $c(\pi)$ by applying c vertex per vertex. A *winning condition* for a player is a set of infinite sequences of colours $W \subset A^\omega$. The winning condition for the other player is $A^\omega \setminus W$. A *game* is a pair (A, W) where A is an arena and W a winning condition for player 1.

Strategy A *strategy* gives the next move of a player given all the previous moves. A strategy for player 1 is $\xi_1 : V^* \times V_1 \to V$ such that for every play $\pi \in V^*$ and a vertex $v \in V_i$, $E(v, \xi_i(\pi, v))$.

Given a *game* $\mathcal{G}$, a starting vertex v_0, and strategies ξ_1 and ξ_2 for the two players, there is an unique play $\rho_{\xi_1,\xi_2}(v_0)$ called *outcome* such that $c(\rho_{\xi_1,\xi_2}(v_0)) \subset W$, *i.e.* is winning. A strategy ξ_1 is *winning* if for all ξ_2, $\rho_{\xi_1,\xi_2}(v_0)$ is winning. Player 1 wins the game from v_0 if she has a winning strategy from v_0 and we note $\mathcal{W}_1$ the set of vertices from which player 1 wins.

Schedulability game Let $P = (P, T, D, Addr, a_0, ins, next)$ an IR program.

We consider the *bounded schedulability problem*: a configuration c is *k-bounded* if the size of the trigger queue of c is bounded by $k \in \mathbb{N}$. We also suppose that variables take values in finite sets.

Our players are the scheduler and the environment: player 1 is the scheduler (it chooses tasks to run on the CPU during time elapses) and player 2 is the environment and chooses environment events, and also executes the IR instructions.

We use two colours: *safe* and *unsafe*. A vertex is coloured as *safe* if the configuration is *time-safe*.

A *clock abstraction* $[c]$ is associated to each configuration $(t, n, s, a, A_s, \mathcal{T}, Tasks)$. We obtain $[c]$ by removing the clocks from the variable in s and we replace each meta-trigger (G, s') in $\mathcal{T}$ by $(G, \sigma_{s'}(p_c) - \sigma_s(p_c))$. It means that we only monitor the duration in physical time since the trigger was activated. A configuration c conforms with w if $\Delta \leq w_t$ for each $(t, s, \Delta) \in Tasks$.

We note $C_{P,w,k}$ the set of clock abstractions. The size of $C_{P,w,k}$ is bounded by $\gamma^{|P|} \cdot |Addr| \cdot (|Tasks| \cdot \gamma^{|P|} \cdot \alpha) \cdot (|\mathcal{T}| \cdot |Addr| \cdot \beta \cdot k)$ where $\alpha = \max\{w_t \mid t \in Tasks\}$, β is the greatest enabling time for all triggers in $\mathcal{T}$, and γ is the greatest size for the sets in which variables take a value.

The vertices of the schedulability game are $V = C_{P,w,k} \times \{1, 2\}$.

Hardness The bounded schedulability problem is hard for EXPTIME. A proof that adapts easily to our game can be found in [20].

Generating a static scheduling strategy A *winning strategy* for player 1 (the scheduler) gives a static strategy that could be used to schedule Antescofo programs in practice. Giotto programs can be compiled to S code, a schedule carrying code [21] for a scheduling machine.

S code adds to the S machine the following instructions to the instructions `schedule(t)` and `call(d)` of the E Machine:

`dispatch(t, g)` dispatches t to execute until t completes, or trigger g is activated.

`idle(g)` idles the S machine until trigger g is activated

`fork`**(a)** executes the code at address a in parallel of code at current address

S code determines the order in which task executes. If S code dispatches only a single task at a time, a system scheduler is not necessary anymore.

Conclusion

Scheduling for real-time multimedia systems raises specific challenges that require particular attention. Examples are triggering and coordination of heterogeneous tasks, especially in Antescofo that uses a physical time, but also a musical time that depends on a particular performance, and how tasks that deal with audio processing interact with control tasks. Moreover, IMS have to ensure a timed scenario, for instance specified in an augmented score, and current IMS do not deal with their reliability and predictability.

We have presented how to formally interleave audio processing with control by using *buffer types* that represent audio buffers and the way of interrupting computations that occur on them, and how to check the property of *time-safety* of IMS timed scenarios, in particular Antescofo augmented scores, based on the extension of an intermediate representation similar to the E code of

the realtime embedded programming language Giotto, and on static analysis procedures run on the graph of the intermediate representation.

However, we cannot currently handle all the possibilities of Antescofo programming. For instance, dealing with loops that have a dynamic period, as well as dynamic tempi or the `whenever` instruction remains to be investigated. Abstract interpretation could also be useful to reduce the size of the graph of the intermediate graph after unfolding.

Another crucial issue of this static analysis approach is the use of worst case execution times. Indeed, WCETs are very difficult to determine on mainstream operating systems such as Linux, Windows, or Mac OS X, because of cache memories and non real-time scheduling at the OS level, which makes any estimation of worst case dubious. However, finding a mean time execution time seems to be more realistic. We would be able able to perform the analysis with the mean execution time, and use dynamic strategies to adapt in case of overload, as described in section 1.4. In Antescofo, composers can precise how they handle missed events by adding `local` and `global` keywords, that can entail non-execution or execution for a reaction to an absent event, and these strategies could be used also to cancel reactions in case of overload.

We also want to explore furthermore the relations between our scheduling test of Part III and the buffer types of Part II: buffer types and TTSDF can be used to generate a schedule for audio processing tasks and this schedule for audio tasks which finally uses a periodic audio callback interacts with control tasks.

We also have started to port [39] Antescofo on Udoo, a small-card nano-computer.

We think that these results can help the music community to use more reliable and more predictable systems, so that what happens at rehearsal time reflects what will happen during a performance, and so that the number of rehearsals to set up IMS correctly decreases.

References

[1] P. Arumí Albó. *Real-time multimedia on off-the-shelf operating systems: from timeliness dataflow models to pattern languages*. PhD thesis, Universitat Pompeu Fabra, 2009.

[2] Ross Bencina. *The SuperCollider Book*, chapter Inside Scynth. MIT Press, 2011.

[3] Gérard Berry. *The Esterel v5 language primer: version v5_91*. Centre de mathématiques appliquées, Ecole des Mines and INRIA, 2000.

[4] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[5] Houssine Chetto, Maryline Silly, and T Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, 1990.

[6] Arshia Cont. A coupled duration-focused architecture for real-time music-to-score alignment. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 32(6):974–987, 2010.

[7] Francis Cottet, Joëlle Delacroix, Claude Kaiser, and Zoubir Mammeri. Scheduling in real-time systems. *Computing Reviews*, 45(12):765, 2004.

[8] François Déchelle, Riccardo Borghesi, Maurizio De Cecco, Enzo Maggi, Butch Rovan, and Norbert Schnell. jMax: an environment for real-time musical applications. *Computer Music Journal*, 23(3):50–58, 1999.

[9] José Echeveste. Un langage temps réel dynamique pour scénariser l'interaction musicien-machine. *TSI. Technique et science informatiques*, 33(7-8):587–626, 2014.

[10] José Echeveste, Arshia Cont, Jean-Louis Giavitto, and Florent Jacquemard. Formalisation des relations temporelles dans un contexte d'accompagnement musical automatique. In *8e Colloque sur la Modélisation des Systèmes Réactifs (MSR'11)*, volume 45, pages 109–124, 2011.

[11] José Echeveste, Arshia Cont, Jean-Louis Giavitto, and Florent Jacquemard. Operational semantics of a domain specific language for real time musician-computer interaction. *Discrete Event Dynamic Systems*, 23(4):343–383, 2013.

[12] José Echeveste, Jean-Louis Giavitto, and Arshia Cont. A Dynamic Timed-Language for Computer-Human Musical Interaction. Research Report RR-8422, December 2013.

[13] José Echeveste. *Un langage de programmation pour composer l'interaction musicale*. PhD thesis, Paris VI, 2015.

[14] Arkadeb Ghosal, Thomas A Henzinger, Christoph M Kirsch, and Marco AA Sanvido. Event-driven programming with logical execution times. In *Hybrid Systems: Computation and Control*, pages 357–371. Springer, 2004.

[15] Jean-Louis Giavitto. Embedding native DSP processing in Antescofo. 2015.

[16] Gérard Grisey. Tempus ex machina: A composer's reflections on musical time. *Contemporary music review*, 2(1):239–275, 1987.

[17] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[18] Thomas A Henzinger, Benjamin Horowitz, and Christoph M Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.

[19] Thomas A Henzinger and Christoph M Kirsch. The embedded machine: Predictable, portable real-time code. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(6):33, 2007.

[20] Thomas A Henzinger, Christoph M Kirsch, Rupak Majumdar, and Slobodan Matic. Time-safety checking for embedded programs. In *Embedded Software*, pages 76–92. Springer, 2002.

[21] Thomas A Henzinger, Christoph M Kirsch, and Slobodan Matic. Schedule-carrying code. In *International Conference on Embedded Software (EMSOFT)*, volume 2855 of *LNCS*, pages 241–256. Springer, 2003.

[22] Christoph M Kirsch. Principles of real-time programming. In *Embedded Software*, pages 61–75. Springer, 2002.

[23] Christoph M Kirsch and Ana Sokolova. The logical execution time paradigm. In *Advances in Real-Time Systems*, pages 103–120. Springer, 2012.

[24] Edward Ashford Lee and Sanjit Arunkumar Seshia. *Introduction to embedded systems: A cyber-physical systems approach*. Lee & Seshia, 2011.

[25] Edward Hashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *Computers, IEEE Transactions on*, 100(1):24–35, 1987.

[26] Stéphane Letz, Dominique Fober, and Yann Orlarey. Comment embarquer le compilateur faust dans vos applications? In *Journées d'Informatique Musicale*, pages 137–140, 2013.

[27] Stéphane Letz, Yann Orlarey, and Dominique Fober. Work stealing scheduler for automatic parallelization in faust. In *Proceedings of the Linux Audio Conference*, 2010.

[28] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[29] Louis Mandel and Marc Pouzet. ReactiveML: a reactive extension to ML. In *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 82–93. ACM, 2005.

[30] Y. Orlarey, D. Fober, and S. Letz. Faust: an efficient functional approach to dsp programming. *New Computational Paradigms for Computer Music*, 2009.

[31] Florence Plateau. *Modèle n-synchrone pour la programmation de réseaux de Kahn à mémoire bornée*. PhD thesis, Paris 11, 2010.

[32] Clement Poncelet and Florent Jacquemard. Compilation of the intermediate representation v1. Technical report, 2015.

[33] Clément Poncelet and Florent Jacquemard. Model based testing of an interactive music system. In *ACM Symposium on Applied Computing (ACM-SAC)*, track SVT, 2015.

[34] Claudius Ptolemaeus. *System Design, Modeling, and Simulation: Using Ptolemy II*. Ptolemy. org, 2014.

[35] M. Puckette. Pure data. In *Proc. Int. Computer Music Conf.*, pages 224–227, Thessaloniki, Greece, September 1997.

[36] Miller Puckette. The Patcher. In *Proceedings of International Computer Music Conference (ICMC)*, pages 420–429, 1988.

[37] Miller Puckette. Max at seventeen. *Comput. Music J.*, 26(4):31–43, 2002.

[38] Robert Rowe. *Machine Musicianship*. MIT Press, Cambridge, MA, USA, 2004.

[39] Nicolás Schmidt Gubbins. First steps towards embedding real-time audio computing in Antescofo. 2015.

[40] Andrew Sorensen and Henry Gardner. Programming with time: cyber-physical programming with impromptu. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '10, pages 822–834, New York, NY, USA, 2010. ACM.

[41] Ralf Steinmetz. Human perception of jitter and media synchronization. *Selected Areas in Communications, IEEE Journal on*, 14(1):61–72, 1996.

[42] Ralf Steinmetz and Klara Nahrstedt. *Multimedia systems*. Springer Science & Business Media, 2004.

[43] Barry Vercoe. *Csound, A Manual for the Audio Processing System and Supporting Programs with Tutorials*. 1993, MIT Media Labs, Ma, 1993.

[44] G. Wang. *The ChucK audio programming language." A strongly-timed and on-the-fly environ/mentality"*. PhD thesis, Princeton University, 2009.

[45] David Zicarelli. How I learned to love a program that does nothing. *Comput. Music J.*, 26(4):44–51, 2002.