A LAZY REAL-TIME SYSTEM ARCHITECTURE FOR INTERACTIVE MUSIC

David Janin

▶ **To cite this version:**

David Janin. A LAZY REAL-TIME SYSTEM ARCHITECTURE FOR INTERACTIVE MUSIC. Journées d'Informatique Musicale, 2012, Mons, France. hal-03041785

HAL Id: hal-03041785
https://hal.science/hal-03041785

Submitted on 5 Dec 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A LAZY REAL-TIME SYSTEM ARCHITECTURE FOR INTERACTIVE MUSIC

David Janin
Université de Bordeaux, LaBRI UMR 5800,
351, cours de la libération,
F-33405 Talence
janin@labri.fr

ABSTRACT

Designing systems that function both in real-time and are interactive, characteristics commonly encountered in computational music, is a challenging task indeed. It becomes even more difficult if we require these systems to be generic with respect to the underlying interactive scores that are to be followed.

The aim of this paper is to define a generic system architecture of this type. Our proposal is based on a lazy real-time kernel that manages both scheduled synchronous events and unpredictable asynchronous inputs in reactive fashion.

This computation by need approach contrasts with standard real-time architectures where the real time kernel is built upon an active periodic loop. It also allows for a clear distinction to be established between interactive music programs written in symbolic time, and interactive music performance executed in real time.

1. INTRODUCTION

Designing musical systems that function both in real-time and are interactive is a challenging task.

At the lowest level, real-time requirements induce a *synchronous* slicing of time with a period defined by a *fixed time quantum*. Computations are limited, for they need to be performed at a very quick pace. For instance, inputs are audio streams possibly filtered by analyzers while a fixed data-flow diagram produces outputs from inputs [4, 7]. Contrary to this, at the interaction management level, some processes are guarded by the advent of external *asynchronous* events. In that case, these data-flow diagrams can be dynamically restructured when event arrived.

In other words, at the lowest level, interactive music systems synchronously compute *sound values*. At the highest level, interactive music systems asynchronously compute *time structures* upon which music itself is built. Even if systems delegate the actual sound production to subcomponents, there is still a need for low level synchronous management of these delegations.

This distinction between low level real-time computations and high-level interactions also appears in the underlying time scale they are based upon. At the lowest level, the *time quantum* is generally measured in 10^{-5}th of a second, e.g. at a $44kHz$ sampling rate. At the interactional level, the *musical tempo* is measured in 10^{-1}th of a second, e.g. from 30 to 300 beats per minute. In between, the expected *reaction* or *precision time* of the system after an asynchronous event is measured in 10^{-3}th of a second, i.e. the lower limit beneath which standard human perception no longer discerns the difference in beat positions.

This shows that mixing real-time and interactional requirements requires a clear distinction to be made between, on the one hand, high level *interactive music controls* governed by a *symbolic time progression* in the underlying *interactive score* and, on the other hand, low level *music production* based on the ticking of a *real time clock*.

Merging these two levels when designing a system will probably give rise to design Flaws. The obtained software will probably be non modular and non reusable. Still, these two levels of design must function hand in hand. The aim of the system architecture explored in this paper is to provide a clear framework for a partnership of this sort.

Main contribution

In this paper, we aim at proposing an abstract generic system architecture for interactive real-time music performance that will fulfill all of the above requirements. At the system architecture level, the central question we address is how the various components in interactive music software will interact.

Our proposal is based on a lazy real-time kernel that handles, in a reactive way, both scheduled synchronous events and unpredictable asynchronous inputs. This contrasts with standard real-time architecture built upon periodic reactive loops.

The resulting real-time and interactive system architecture is peculiarly robust with respect to occasional time drifts : whenever forced out of time, the running system will auto-stabilize on time as if no time drift had ever occurred.

Within the specific framework of music and the architecture of the system thereby established, we also encode a clear distinction between symbolic time (beats) and real-time (seconds), each being related to the other via a constantly changing tempo.

As a consequence, despite its simplicity, this model induces several layers : from low-level input/output controllers to high-level music controllers, each with its own clear and distinct functional specifications. In particular, the music controller layer may simply be seen as an abstract interpreter of (arbitrary) symbolic interactive scores. This guarantees genericity.

Related works and subject position in the field

The last decades have seen the development of various software programs for Computer Assisted Music either used on stage for live performances or integral to multimedia applications for rich interactive audio supports. These softwares range from low level sound synthesis and control tools such as *Faust* [7] or *Max/MSP* [4], to high level composition assistants such as *Elody* [15] or *OpenMusic* [1] to name but a few.

However, the design and execution of computer assisted interactive music still remains a challenging task. We first need to gain a better understanding of the way low level(synchronous) sound synthesis and control may be combined with high level (asynchronous) musical inputs. It some sense, there is a increasing need for mixed systems that provide high level interactive control structures for the description of potentially complex interactions between lower level sound or music features.

Many systems of this sort, see [5] among others, can be seen as forms of *domain specific languages (DSL)* that are adapted to the design and implementation of interactive scores. Do these languages attain a sufficient level of abstraction ? Do they induce an adequate notion of interactive scores ? The pragmatic reuse of existing and reliable low level tools somehow messes up the picture.

There is as yet no appropriate yardstick for measuring the expressivity of interactive music description. Even more importantly, the temporal and spatial means of structuring interactive scores, thereby aiding composers in this highly arduous task, still need to be better understood and developed.

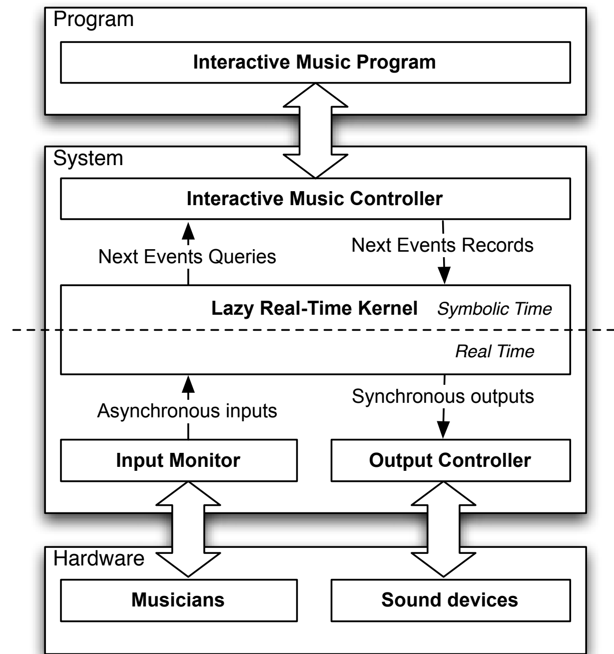
This paper, without forasmuch being able to clearly define the exact *nature* of an interactive musical score, aims at establishing a more precise understanding of *where* and *how* such interactive scores may be played. As a result, we may develop an abstract operational semantics for such scores that has some similarity with Alur and Dill timed automata [3] or, more generally, hybrid systems [11].

This enforces the general and well-accepted idea that known models, developed for years in the presumably distinct application context of critical embedded systems, may nonetheless be adapted efficiently to the application context of interactive music.

2. GENERAL ARCHITECTURE

The main purpose of a system is to bridge the gap between the program layer and a hardware layer. Our proposal, oriented towards interactive music performance, follows this general specification.

Its main components and connection with the environment are depicted in the diagram below.



The system itself is composed of three layers of components whose functionality may be described with more details.

The interactive music controller. This component acts as an interactive partition (or interactive music program) follower ; it receives queries from the musical events in the real-time kernel with symbolic time stamps and produces back to the real-time kernel, in coherence with the score followed and ? ? the record of musical events to be performed at the next relevant symbolic time value.

Lazy real-time kernel. This component handles the lazy real-time loop (described below in detail), a call by need real-time scheduler ; functionally, it handles the communication between the symbolic time layer defined by the music controller and the real time layer defined by both the input monitor and the output controller ;

Input monitor / output controller. These components are in charge of the communication between the interactive music system and the music hardware :

(a) the input subcomponent receives asynchronous input events from musical instrument users (musicians) and transmits their formatted descriptions to the real-time kernel with no delay,

(b) the output subcomponent produces musical streams upon reception of their formatted descriptions received from the real-time kernel.

3. LAZY REAL TIME LOOP

The lazy real time loop, running in the real-time kernel, is essentially in charge of the reactive communication between the music controller denoted `M`, the input monitor denoted by `I` and the output controller denoted by `O`.

It is described below in an object-oriented syntax.

```
Event * E; // eventsList
Time * T;  // next firing date
T = M.getNextEventDate();
While (T.isDefined()) do
{
  Event * E = I.WaitEventUntil(T);
  if (E.isDefined())
     // Received event case
     E = M.updateReceiEvent(E,Now);
  else
     // Planed event case
     E = M.getNextEventAtTime(T);
  O.fireEventAtTime(E);
  T = M.getNextEventDate();
}
```

This loop structure calls for some explanation. As opposed to standard real time architectures, it is built upon a mechanism involving a lazy reactive evaluation schema triggered by *two competing events* :

(1) an *unpredictable asynchronous event* `E` is received from the input monitor `I` ; in that case, the event is passed with no delay to the music controller that sends back a possibly enriched event description that is fired immediately ; such an event is called a *received event*,

(2) a *scheduled next event date* expires and the new scheduled synchronous event `E`, provided by the music controller `M`, is passed to the output controller to be fired at time `T` ; such an event is called *scheduled* or *planned events*.

The monitoring of these two competing events is implemented via the call of `WaitEventUntil(T)` on the input monitor `I`. This method returns the undefined object `nil` when no received event has occurred and the current date is greater [1] or equal to the next event date encoded in `T`. We describe how these events are managed in further detail below.

Management of a received event. By default, when an unpredictable event `E` is received, a copycat scenario takes place. This scenario is implemented as follows. Method `updateReceiEvent(E,Now)` sends back the event `E`. The next event scheduled date, returned by the next call to `getNextEventDate(T)`, remains unchanged.

In all cases, the received event is passed to the music controller which may then, upon reception, update its own control states. The event actually fired, sent back by the controller, may even be different to the received event. It may have been enriched. It may even be ignored when the music controller sends it back `nil`.

More precisely, the musical consequence of the reception of an asynchronous unpredictable event, that depends both on its reception date and its value, is governed by the interactive score followed by the music controller. For instance, it may even be the case that the next planned event date changes after updating by the music controller. This generally happens when a received event induces a change of tempo.

Management of a planned event. When a scheduled event date expires, the music controller is asked for the event `E` to be fired. This is done by the lazy real-time kernel, using the `getNextEventAtTime(T)` method. By default, it simply reads the music score, sends back the next event, and updates its own record of the next scheduled event date.

Until that firing date, there is no need to know which event is to be fired. It follows that this event may be simply *computed* by the music controller, when asked via the `getNextEventAtTime(T)`. This means that the music score can be truly interactive, in the sense that, at any time, the played event may depend on the history of the musical events that have been received and produced so far.

At any time, the next musical event to be played depends on the current internal state of the interactive music score. This programmatic feature of the score is discussed a little further in Section 5 below.

Firing events on time. In all cases, both (enriched) received events or planned events are sent to the output monitor to be fired immediately. In practical implementations of this process, we make the firing of a planned event a little more subtle.

More precisely, method `WaitEventUntil(T)` resumes some `delta` seconds *before* the real time scheduled date expires. This anticipates the amount of time needed for the computation of the next event `E`. Firing a planned event is thereby performed as follows :

(1) if the current date is sooner than the scheduled firing date, the real-time kernel waits during the remaining lapse of time ; the event is then *fired just on time*, i.e. this is the expected default case,

(2) if the current date is equal or `gamma` seconds later than the scheduled firing date, the event is immediately fired *almost on time*, i.e. `gamma` is the allowed lapse of time for an error in precision,

(3) if the current date is greater than the scheduled firing date plus `gamma`, this means the system is late ; the firing of the event can be omitted in order to avoid parasite noises which are out of time ; it is expected that the system will auto-stabilize.

Parameters `delta` and `gamme` may be set adequately according to the performance of the computer the system is running on.

1 . this may happend when the all system is late

4. REAL TIME VS SYMBOLIC TIME MANAGEMENT

One of the major tasks of the lazy real-time kernel is to ensure the conversion from real time, handled by the input monitor and the output controller, to symbolic time handled by the interactive music controller.

In the lazy real-time loop described above, we have concealed the means by which real-time dates are converted into symbolic dates and vice-versa, by which symbolic dates are converted into real-time dates.

The methods and attributes of class `Time` manage these conversions operating back and forth.

4.1. Real and symbolic current time handling

The first basic attributes of the class `Time` are :

(1) `SymbCurrentD` defined as the *symbolic current date*, i.e. the (float) number of *symbolic time units* (or *beats*) elapsed since the beginning of the musical performance until *now*,

(2) `RealCurrentD` defined as the *real-time current date*, i.e. the (float) number of *real time units* (or *minutes*) elapsed since the beginning of the musical performance until *now*,

(3) `tempo` defined as the evolving speed of the symbolic date w.r.t. the real-time date (in *beats per minute*).

Observe that at any time, the value of the *real-time current date* is defined while the value of the *symbolic current date* needs to be computed.

In the simplest case, when the *tempo* is constant, the following *invariant property* holds.

```
SymbCurrentD == tempo * RealCurrentD;
```

This shows how the *symbolic current date* can be computed from the (constant) value of the *tempo* and the *real-time current date*.

In the general case, when the *tempo* is not constant, things are a little more complex. In this light the following hypothesis may be posited :

> (H) Between any two successive events, be they received or planned events, *tempo* is constant.

May we therefore argue that this hypothesis is a constraint ? We may observe that any change of tempo [2] can be modeled as an event in its own right and that therefore this hypothesis is simply a modeling choice.

On the basis of this simple hypothesis, managing the symbolic vs real-time conversion may be achieved by recording the *last* values of symbolic or real-time dates in two extra attributes. More formally, these extra attributes of the class `Time` are :

(4) `SymbLastD` defined to be the *symbolic last event date*,

(5) `RealLastD` defined to be the *real-time last event date*.

The *invariant property* associated with these new attributes is defined as follows :

```
SymbCurrentD == SymbLastD +
    tempo * (RealCurrentD - RealLastD);
```

This shows, in the general case, how the value of the *symbolic current date* may be computed from the value of the *real-time current date*.

It may be observed that this equation is essentially needed when updating the music controller's current state upon the reception of an event `E`. Indeed, the music controller `M` only handles symbolic time in the interactive music score. Converting the real-time date of reception of a given event `E` into its corresponding symbolic time value is thus a necessity.

4.2. Scheduled dates updates

In the lazy loop described above there is yet another notion of time which needs to be modeled : namely, the *scheduled symbolic date* and *scheduled real-time date* of the next planned event.

This is done by using two more attributes in the class *Time* which are :

(6) `SymbSchedD` defined as the *symbolic scheduled date* of the next planned event,

(7) `RealSchedD` defined as the *real-time scheduled date* of the next planned event.

These scheduled dates are related with last dates in the same way as current dates are related with last dates. However, the *symbolic scheduled date* is *provided* by the music controller when computing the next event date. It follows that we now need to compute the *real-time scheduled date* from that symbolic value.

The relevant *invariant property* is thus defined as follows.

```
// whenever needed
RealSchedD = RealLastD +
    (SymbSchedD - SymbLastD)/tempo;
```

This shows how the value of the *real-time scheduled date* is computed from the value of the *symbolic scheduled date*.

This equation is essentially required when the input monitor `I` is waiting for a input event prior to some scheduled date `T`. Since monitor `I` only handles real-time dates, the symbolic scheduled date provided by the music controller will necessarily have to be converted.

4.3. Last date updates

We are now ready to describe the update procedure of the *real-time last event date* and the *symbolic last event date*. By definition, these dates must be updated every time an event is fired.

2 . either from the input monitor or from the music controller

At first sight, intuition might lead us to intuit that these updates values are to be calculated with the values of the *real-time* and the *symbolic current date* of the given event production. Yet this intuition would indeed be wrong. By definition, the *real-time current date* changes all the time. It may even be the case that *real-time current date* is greater than *real-time scheduled date* since firing an event also takes a certain amount of time. Even worse, it may be the case that the *real-time current date* values changes from the moment we *want* to read its value from the moment we actually *ascertain* its value.

It transpires that these updates must be computed with the values of the *real-time* and *symbolic scheduled dates* of the event that has just been performed. In other words, we actually perform the following update :

```
RealLastD = RealSchedD;
SymbLastD = SymbSchedD;
```

The updating of the tempo, an update that may be associated with the event fired, occurs immediately after the last date updates :

```
tempo = E.newTempo();
```

In order to increase the robustness of the code, the new tempo, associated with any event, may be set, by default, in accordance with the current tempo value.

4.4. Properties defining the robustness of the lazy time handling

This handling of symbolic and real-time dates enjoys a number of key properties which are worthy of discussion.

Time precision. With this architecture, scheduled dates are *computed* from previous scheduled dates and tempo at any moment in the run of an interactive score. It follows that the time precision may be measured, say, just before firing an event, as follows :

```
timePrecision =
        RealCurrentD - RealSchedD;
```

which is positive when the firing of the event occurs *after* the scheduled date.

Experiments on a prototype implementation of that system in *ObjectiveC* under *MacOSX* shows that this time precision just remains below a few ms which is just enough for musical performance.

Robustness w.r.t. time drifting. When firing an event, if the time precision is too great, then we may seek to avoid the sound resulting from this event being produced. It results that this is easily implemented by simply *guarding* the actual firing of an event by a comparison between the measured time precision and the maximal allowed one.

In doing so, the resulting system becomes remarkably robust : if the system is paused for some reason (either intentionally or because of an overload of the computer running the system) then, upon resuming, the system not only omits to play the outdated events, but also, since the scheduled dates are computed data, the system runs forward through the score until it reaches the correct symbolic date corresponding to the actual real-time date.

In other words, after a pause, the system resumes as if *no pause had ever occurred* ! In a live performance context, especially when real musicians continue to play while the system is paused, or when listeners are dancing or even just finger tapping, the fact that the system will resume on time is a particularly desirable property.

We may observe that this property is not satisfied by standard *streaming software* since, quite often, audio or video frames are not time stamped.

5. INTERACTIVE MUSIC SCORES

At this point, the real-time kernel of the proposed system requires further analysis. As the input monitor and output controller are rather simple at that level of abstraction, it remains for us to describe the way the interactive music controller can be *programmed* in greater depth.

To some extent, the interactive music controller is a symbolic execution layer upon which an interactive music specification, no longer seen just as a score to be followed, is run. In our approach : *an interactive score is defined as a timed reactive program that produces the musical score on line, event after event, in step with the history of the received input events.*

In this section, the characteristic of such musical programs will be described in greater depth. It is not our intention to defend a given *syntax* for these programs. We are more concerned by the operational *semantic* features these programs may have.

5.1. Some basic musical programs

In order to better intuit how such timed interactive programs may be defined, we describe below several typical musical scenarios and show how they may be encoded.

Immediate start. The first start scenario envisaged arises when we want the music to be started immediately upon activation of the system.

This can be done by a controller that sets the initial next scheduled event date to zero. Indeed, in doing so, the real time kernel immediately prompts the music controller for the first musical event to be performed.

Conditional start on input. Contrary to this, another possible start scenario arises when we want the music to be started by an external input event that may occur after an unpredictable delay.

In turn, this may simply be achieved by a controller that sets the initial next scheduled event date to infinity ($+\infty$). This way, the system will necessarily wait for an external event.

Observe that if such an infinite date value is not available, this can still be done by repeatedly producing a silent scheduled event until the first external event is received.

End scenario. We may also ask how such a system may be stopped. Might we therefore argue that the architecture described here engenders never ending musical pieces ? Actually, the lazy loop makes this quite clear. The music controller stops the system by simply sending an undefined (or *nil*) next event date. In other words, this undefined date acts as the final bar of an interactive score.

Play through metronome scenario with varying tempo. Finally, a metronome with play through capacity is also easily encoded as a music controller.

Indeed, repeatedly, the symbolic date of the next scheduled event is by default increased at every beat by one : the metronome is expected to tick at every beat. At any other date, upon reception of an external event `E`, the default behavior described in Section 3 is executed, i.e. method `updateReceiEvent(E,Now)` simply sends back the event `E`.

In doing so, a simple additional input interface with a tempo change cursor and a start/stop button may complete the picture in order to produce the missing start/stop and tempo change events.

5.2. Music programs as symbolic timed automata

At any scheduled date, the interactive music controller essentially provides the next scheduled event to be fired. Of course this event must be known before being fired. However, until its firing date, any asynchronous external event may occur and change this characteristic. This means that the next scheduled event must be computed right on time when needed for firing : this computation by need is the consummate definition of lazy computation.

But what about interactive scores ? The proposed architecture permits the programming of timed controllers in an almost pure symbolic time setting. The real time interpretation is solely governed by the evolving variable `tempo`. Indeed, this tempo may be changed either by hand with adhoc input events or programmatically by special control events from the symbolic controller (see the end of Section 4.3).

This means that the interactive music controller behaves like a sort of input/output timed automata [3] interpreter. Reading a timed input event updates the state of the running automaton (the automaton is reactive). After some delay, depending on the active state, a default transition is always activated by sending back a timed planned event to the system (the automaton is time active).

What exact type of timed automata are to be executed by the music controller layer ? This is still a matter of research. Our proposal provides some indication of *how* to run an interactive score. The true nature of an interactive score is still an open question.

6. CONCLUSION

For an effective use of this proposed system we now need to gain a deeper understanding of how the music controller can be programmed. At the operational level, music controllers look like timed automata. But this fairly low level model seems inadequate for interactive music composition.

There is still a need to develop a high level modular language for the description of interactive scores. This might be achieved by pursuing the research which led to proposals such as *iScore* [2]. In particular, we may consider the following three complementary research projects.

The first concerns the capacity of such a score to describe musical anticipation in a simple fashion as one of the main conceptual tools used in music composition.

Already in the 80's some proposals emerged in this direction [6]. But there are still many questions to be answered. Aside statistical analysis and continuation techniques that are proposed by softwares such Continuator [16] or OMax [9], we also believe that structural analysis of musical languages may be conducted. For instance, musical anticipation may be envisaged, on a more abstract level than music scores, as a generalization of musical anacrusis [12].

The second concerns the various combinations possible of interactive programs which may be defined. The sequential and parallel composition of elements of interactive scores are obviously required. But what type of sequential composition ? What type of parallel composition ? How may input events be distributed among the different elements of these scores ? Are they to be duplicated ? Buffered ? An initial study of sequential composition, both from the perspective of music modeling [12] or from a purely theoretical point view [13, 14] already shows that, together with anticipation modeling, a lot remains to be said.

The third concerns the hierarchical description of the music. It seems that composers are looking for ways of thinking about their music on several levels of abstraction : say from elementary sounds to performance in a concert via musical motifs, movements, pieces, etc...

Hierarchical system modeling techniques have already been defined in various areas in computer science. In particular, statecharts in UML [10] is based on a hierarchical description of this type. However, standard statecharts semantics may need to be adapted for hierarchical interactive music descriptions.

These ideas for future research, on the musical side of our proposal, also need to be combined with existing techniques and concepts for low level audio stream analysis (as inputs) or audio stream production (as outputs) or with those still to be developed.

Synchronizing two elements from scores that result from real-time computation issuing from the history of the global piece being performed is one thing. Combining the associated audio streams these elements realize is quite

another. It seems that each operator defined on the symbolic side of music demands a counterpart on the realization side.

In all cases, we expect that the present proposed system architecture will facilitate further experimentation.

7. REFERENCES

[1] Bresson J. Agon C. and Assayag G. *The OM composer's Book, Vol.1 & Vol.2*. Collection Musique/Sciences. Ircam/Delatour, 2006.

[2] Antoine Allombert, Myriam Desainte-Catherine, and Gérard Assayag. Iscore : a system for writing interaction. In *Third International Conference on Digital Interactive Media in Entertainment and Arts (DIMEA 2008)*, pages 360–367. ACM, 2008.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994.

[4] Alessandro Cipriani and Maurizio Giri. *Electronic Music and Sound Design - Theory and Practice with Max/Msp*. Contemponet, 2010.

[5] Arshia Cont. Antescofo : Anticipatory synchronization and control of interactive parameters in computer music. In *International Computer Music Conference (ICMC)*, 2008.

[6] P. Desain and H. Honing. Loco : a composition microworld in logo. *Computer Music Journal*, 12(3) :30–42, 1988.

[7] D. Fober, Y. Orlarey, and S. Letz. Faust architectures design and OSC support. In *14th Int. Conference on Digital Audio Effects (DAFx-11)*, pages 231–216. IRCAM, 2011.

[8] Alexandre R. J. François and Elaine Chew. An architectural framework for interactive music systems. In *International Conference on New Interfaces for Musical Expression*, pages 150–155, 2006.

[9] M. Chemillier G. Assayag, G. Bloch. Omax-ofon. In *Sound and Music Computing (SMC) 2006*, 2006.

[10] David Harel. Statecharts in the making : a personal account. In *Proceedings of the Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III), San Diego, California, USA, 9-10 June 2007*, pages 1–43. ACM, 2007.

[11] Thomas A. Henzinger. The theory of hybrid automata. In *Proceedings, 11th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 278–292. IEEE Computer Society, 1996.

[12] David Janin. Modélisation compositionnelle des structures rythmiques : une exploration didactique. Technical Report RR-1455-11, LaBRI, Université de Bordeaux, August 2011.

[13] David Janin. On languages of one-dimensional overlapping tiles. Technical Report RR-1457-12, LaBRI, Université de Bordeaux, January 2012.

[14] David Janin. Quasi-recognizable vs MSO definable languages of one-dimentionnal overlaping tiles. Technical Report RR-1458-12, LaBRI, Université de Bordeaux, February 2012.

[15] S. Letz, Y. Orlarey, and D. Fober. Real-time composition in Elody. In *Proceedings of the International Computer Music Conference*, pages 336–339. ICMA, 2000.

[16] F. Pachet. The continuator : Musical interaction with style. In *Proceedings of ICMC*, pages 211–218, Göteborg, Sweden, September 2002. ICMA. best paper award.