A Lazy Real-Time System Architecture For Interactive Music

David Janin

LaBRI, CNRS UMR 5800
Laboratoire Bordelais de Recherche en Informatique

Rapport de recherche RR-1460-12

A Lazy Real-Time System Architecture For Interactive Music

April 13, 2012

David Janin,
LaBRI, IPB, Université de Bordeaux

Contents

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

April 13, 2012

Abstract

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

In this paper, we aim at defining such a generic system architecture. Our proposal is based on a lazy real-time kernel that manages, in a reactive way, both scheduled synchronous events and unpredictable asynchronous inputs.

This reactive approach contrasts with standard real-time architectures where the real time kernel is built upon an active periodic loop. It also permits a clear distinction between interactive music programs: written in symbolic time, and interactive music performance: executed in real time.

1 Introduction

Designing systems that are both real-time and interactive, features commonly encountered in computational music, 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 fast. For instance, at such a small time scale step, inputs are audio streams possibly filtered by some analyzers and some fixed set of data-flow diagrams produce outputs from inputs [4, 6]. Streams synthesis can even be delegated to specialized subcomponents, be them hardware or software, systems providing then higher level parameters for such synthesis. At this level, systems' behaviors are essentially unstructured.

On the opposite side, at the interaction level, systems' behaviors are structured by means of high level concepts. Some processes are guarded by arrivals

of external events: *asynchronous* events that may change the structure of the system behavior itself. In particular, data-flow diagrams that produce outputs from inputs can be dynamically changed at that level.

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 of 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 $44kHz$ of audio sampling rate. At the interaction 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 limit under which standard human perception no longer make the difference in beat positions.

This tells us that mixing real-time and interaction requirements requires a clear distinction 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 a *real time clock*.

Merging these two levels in a design will probably lead to design flaws with non modular and non reusable resulting software. Still, these two levels of design must collaborate. This is the purpose of our proposed system architecture.

Main contribution

In this paper, we aim at proposing an abstract generic system architecture for interactive real-time music performance that will fulfill all the above requirements. Raised at the system architecture level, the central question we adress is how the various components in an 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 a periodic reactive loops.

The resulting real-time and interactive system architecture turns out to be peculiarly robust w.r.t. occasional time drifts: whenever forced out of time, the running system will auto-stabilize on time as if no time drift had ever occur.

More specifically oriented towards music, we also encode, at the architecture level a clear distinction between symbolic time (beats) and real-time (seconds), each being related with the other via a possibly evolving tempo.

As a consequence, despite its simplicity, this model induces several architecture layers: from low-level input/output controllers to high-level music controllers, that have clear and distinct functional specifications. In particular, the

music controller layer can just be seen as an abstract interpreter of (arbitrary) symbolic interactive scores. This guarantees genericity.

Related works and subject position in the field

These last decades have seen the development of many softwares for Computer Assisted Music either used on stage for live performances or used within multimedia applications for rich interactive audio supports. These softwares range from low level sound synthesis and control tools such as *Faust* [6] or *Max/MSP* [4], to high level composition assistants such as *Elody* [14] or *OpenMusic* [1] to mention just a few of them.

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

Many of such systems, see [5] among others as an example, can be seen as sorts of *domain specific languages (DSL)* that are adapted to the design and implementation of interactive scores.

Are these languages reaching a good abstraction level ? Do they induce a rich enough notion of interactive scores ? The pragmatic reuse of existing and reliable low level tools somehow mess up the picture.

There is yet no appropriate expressiveness yardstick for interactive music description. Even worse, temporal and spacial structurations of such interactive scores, highly relevant in order to ease composers' task when writing interactive scores, still need to be better understood and developed.

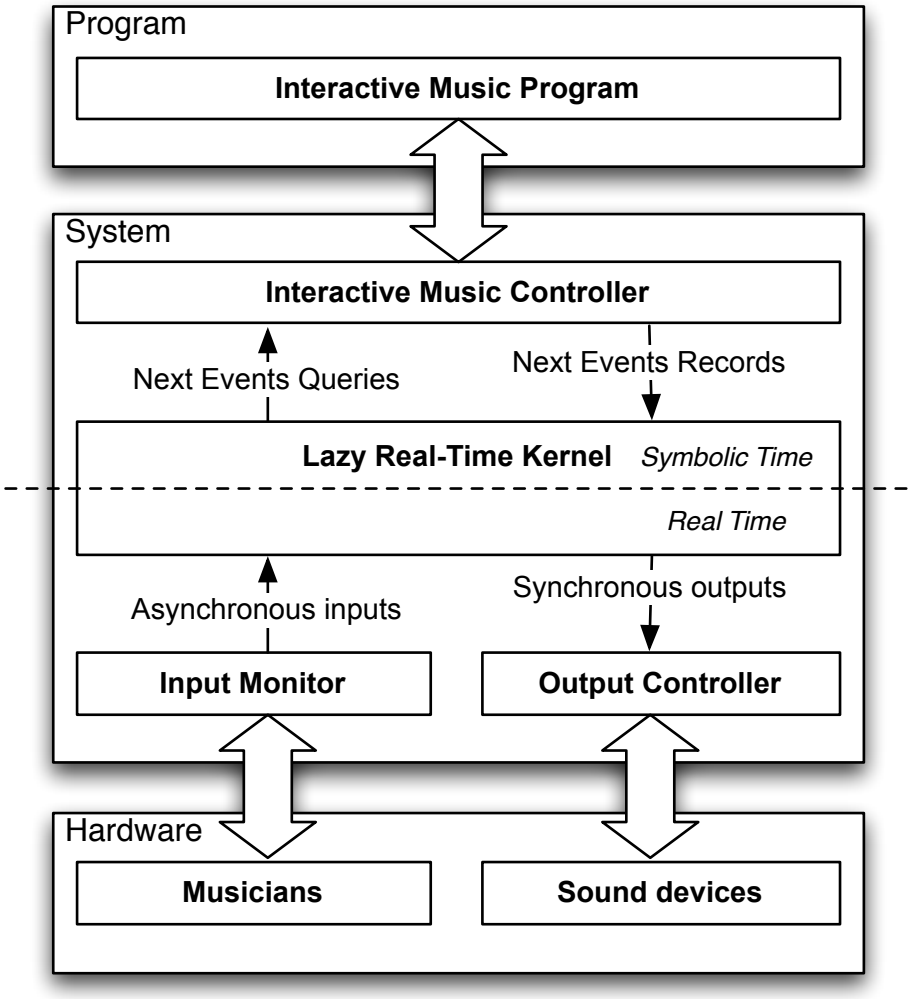
In the present paper, while we do not know *what* exactly is an interactive musical score, we still aim at understanding precisely *where* and *how* such interactive scores can be played. As a result, we obtain an abstract operational semantics for such scores that has some similarity with Alur and Dill timed automata [3] or, more generally, hybrid systems [**?**].

This enforces the general and well admitted idea that known models, developed for years in the presumably distinct application context of critical embedded systems, can still 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 its connection with the environment are depicted in the following picture.



The system itself is composed of three components' layers which functionality can be described as follows.

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

Lazy real-time kernel. This component handles the lazy real-time loop (described below in detail): sort of lazy tough 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 musical hardware:

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

(b) the output subcomponent produces musical streams upon reception of their formated 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 like 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 of 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* is expired 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 had occur and the current date is greater[1] or equal than the next event date encoded in `T`. We describe below a little further how these events are managed.

Management of a received event. By default, when an unpredictable event `E` is received, a copycat scenario can take place. This default scenario is implemented as follows: the `updateReceiEvent(E,Now)` method sends back the event `E` and 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 that can, upon reception, update its own control states. The event actually fired, sent back by the controller, can even be different from the received event. It can have been enriched. It can even be ignored when the music controller send 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 that update of the music controller. This happens in particular 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-ime kernel, calling the `getNextEventAtTime(T)` method. By default, it just read the music score, send back the next event, and update its own record of the next schedule event date.

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

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

[1] this may happend when the all system is late

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 implementations, the firing of planned event can be made a little more subtle.

More precisely, method *WaitEventUntil(T)* may resumes some `delta` seconds *before* the real time scheduled date expires. The computation of the next event `E` can thus be anticipated.

Firing a planned event is then performed as follows:

(1) if the current date is smaller than the scheduled firing date, the real-time kernel wait the remaining 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 greater than the scheduled firing date, the event is immediately fire *almost on time*, i.e. `gamma` is the allowed time precision error,

(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 out of time parasit noise; it is expected that the system will auto-stabilize.

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

4 Real time vs symbolic time management

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

In the lazy real-time loop described above we have just hidden how real-time dates are converted into symbolic dates and vice-versa, how symbolic dates are converted into real-time dates.

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

4.1 Real and symbolic current time handling

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

(1) `SymbCurrentD` defined to be the *symbolic current date*, i.e. the (float) number of *symbolic time units* (say *beats*) elapsed since the beginning of the play till *now*,

(2) `RealCurrentD` defined to be the *real-time current date*, i.e. the (float) number of *real time units* (say *minutes*) elapsed since the beginning of the play till *now*,

(3) `tempo` defined to be the evolving speed of the symbolic date w.r.t. the real-time date (say 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;
```

It tells 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. We make the following hypothesis:

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

Is that hypothesis a constraint ? We can observe that, provided any change of tempo[2] is modeled itself as an event, this hypothesis is just a modeling choice.

Under that simple hypothesis, managing the symbolic vs real-time conversion can just be done 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);
```

It tells, in the general case, how the value of the *symbolic current date* can be computed from the value of the *real-time current date*.

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

[2] either from the input monitor or from the music controller

4.2 Scheduled dates updates

In the lazy loop described above there is yet another notion of time that needs to be modeled : it is 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* that are:

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

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

These scheduled dates are related with last dates in the same way 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 need now 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;
```

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

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

4.3 Last dates 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.

At first sight, intuition may tell that these updates values are to be done with the values of the *real-time* and the *symbolic current date* of that event production. But this intuition is wrong. Indeed, by definition, the *real-time current date* is changing all the time. It may even occur for instance that *real-time current date* is bigger than *real-time scheduled date* since firing an event also takes time. Even worse, it may even 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 *get* its value.

It turns out that these updates are to be done with the values of the *real-time* and *symbolic scheduled dates* of the event that has just been performed. In other words, we rather perform the following update:

```
// right after firing an event
RealLastD = RealSchedD;
SymbLastD = SymbSchedD;
```

The update of the tempo, update that may be associated to the event fired, just occurs right after the last dates updates :

```
// right after last dates updates
tempo = E.newTempo();
```

In order to increase the robustness of the code, this new tempo associated to any event can be set, by default, to the current tempo value.

4.4 Robustness properties of the lazy time handling

This handling of symbolic and real-time dates have several properties that are worth mentioning.

Time precision. With this architecture, at any time in the run of an interactive score, scheduled dates are *computed* from previous scheduled dates and tempo. It follows that the time precision can just be measure, 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 big then we may want to avoid the sound resulting from this event to be produced. It turns out that this is easily implemented by just *guarding* the actual firing of an event by the adequate condition upon the measured time precision.

Doing so, it occurs that the resulting system has a remarquable robustness property: if ever the system is paused (either intentionally or because of an overload of the computer where it is running on) then, upon resume, the system not only omits to play the outdated events, but moreover, since the scheduled dates are computed data, the system runs forward through the score until it reaches the correct symbolic date that correspond to the actual real-time date.

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

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

5 Interactive music scores

At this point, the real-time kernel of the proposed system shall be clearly understood. As the input monitor and output controller are, at that level of abstraction, rather simple, it remains to describe a litte bit further the way the interactive music controller can be *programmed*.

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

In this section, we aim at describing a little further what are the characteristic of such musical programs. We must say we are yet not interested in defending such or such *syntax* for these programs. We are rather concerned by the operational *semantics* features these programs may have.

5.1 Some basic musical programs

In order to give some more intuition on how such timed interactive program can be defined, we describe below several typical (peaces of) musical scenarios and we show how they can be encoded.

Immediate start. The first start scenario one can think of is when we want the music to be started immediately upon activation of the system.

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

Conditional start on input. On the opposite side, another start scenario one can think of is when we want the music to be started by an external input event that may occur after an unpredictable delay.

In turn, this can simply be done 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 just repeatedly producing a silent scheduled event until the first external event is received.

End scenario. We may also ask how such a system can be stopped. Do our architecture forces never ending musical pieces ?

Actually, the lazy loop makes it quite clear. The music controller stops the system just by sending an undefined (or *nil*) next event date. In other words, this undefined date just act as the final bar of an interactive score.

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

Indeed, repeatedly, the next scheduled event symbolic date 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)` just send back the event `E`.

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 changes 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 that can change its characteristic. This means that this next scheduled event must be computed right on time when needed to be fired: this computation by need is precisely the definition of lazy computation.

But what are 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 can 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 behave like sort of a input/output timed automata [3] interpreter. Reading a timed input event updates the running automaton state (the automaton is reactive). After some delay, depending on the active state, a default transition is always activated sending back to the system a timed planned event (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 gives us some hint about *how* to run an interactive score. *What* is an interactive score is still an open question.

6 Conclusion

For an effective use of this proposed system we need now a deeper understanding of how the music controller can be programmed. At the operational level, music controllers look like timed automata. But this model, fairly low level, just 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 can be done pursuing the research that already

led to proposals such as *iScore* [2]. In particular, we may consider the following three complementary research directions.

The first one concerns the capacity of such a score to describe rather simply musical anticipation which is one of the main conceptual tool for music composition.

Already in the 80's some proposals are made in this direction [**?**]. But there are still many questions to be answered. Aside statistical analysis and continuation techniques that are proposed by softwares such Continuator [15] or OMax [8], we also believe that structural analysis of musical languages can still be conducted. For instance, musical anticipation can be seen, at more abstract level than music scores, as a generalization of musical anacrusis [10].

The second one concerns the various combination of interactive program one may define. Sequential and parallel composition of pieces of interactive scores are obviously needed. But what type of sequential composition ? What type of parallel composition ? How input events are to be distributed among these peaces of scores ? Shall they be duplicated ? Buffered ? A first study of sequential composition, both in music modeling perspectives [10] or with purely theoretical point view [11, 13] already shows that, together with anticipation modeling, still a lot can be said.

The third one concerns hierarchical description of the music. It seems that composers are in need of ways to think about their music at several abstraction levels: say from elementary sounds up to concert via musical motifs, mouvements, pieces, etc...

Hierarchical system modeling techniques have already been defined in various areas in computer science. In particular, statecharts in UML [9] is based from such hierarchical description. However, standard statecharts semantics may need to be adapted for hierarchical interactive music descriptions. The music produced by low level components cannot be ended any time a higher level component changes of state.

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

Synchronizing on the fly two pieces of scores that result from real-time computation that depends on the history of the global piece being performed is one thing. Combining on the fly the associated audio streams that realize these pieces of scores is another thing.

It seems that each operators defined on the symbolic side of music will need its counterpart on the realization side.

In all cases, we expect that the present proposed system architecture, actually under development, will ease experimenting these proposals.

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

[9] 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.

[10] 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.

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

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

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

[14] 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.

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