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

March 3, 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

March 3, 2012

Abstract

Designing a system that is both real-time and interactive, features that are commonly encountered in computational music, is a challenging task. It becomes even more difficult if we require the designed system to be not only reliable and maintainable, but also generic in the underlying interactive scores that are to be followed. In this paper, we aim at defining such a generic architecture.

Our proposal is based on a lazy real-time kernel that handles, somehow 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.

As a result, we do obtain a real-time architecture that is robust w.r.t. occasional time drifts. More precisely, whenever forced out of time (say with an external pause) the running system (upon resume) auto-stabilizes back 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.

Despite its simplicity, our proposal induces several architecture layers : real-time, mixed and symbolic, that have clear and distinct functional specifications. In particular, the symbolic musical control layer can just be seen as an abstract interpreter of symbolic interactive scores. In a research context where the notion of interactif scores itself still needs to be better understood, this is a rather good news.

1 Introduction

Designing a system that is both real-time and interactive, features that are commonly encountered in computational music, is a challenging task.

At the lowest level, the real-time requirement induces an apriori *synchronous periodic* slicing of time with a period defined by the expected *time precision*. However, most often at such a small time scale step, no event hence no computation is needed.

On the opposite side, the interaction requirement calls for an *asynchronous aperiodic* availability of maximum computational power. Indeed, each external asynchronous event must be treated as efficiently as possible in order to increase system reactivity.

One may also notice that the elapsed time between two external events, i.e. some factor or fraction of the underlying *musical tempo*, is generally considerably larger than the expected *time precision*.

In other words, at the lowest level, the real-time and the interactive requirements seem to induce opposite efficiency constraints at rather incomparable time scales.

At a more abstract level, the impact of asynchronous real-time external inputs on the symbolically synchronous output flows can be complex.

For instance, a change of rhythmic intention in the inputs may induces a change of the underlying musical tempo. It follows that, while the *rhythmic vocabulary* of the music is left unchanged, the *real-time realization* of the music can just be different.

Shall these two levels of design be merged and the resulting system will have a high probability to be erroneous, non modular, and hardly generic w.r.t. the underlying interactive scores that are performed.

In other words, at the abstract control level, the real-time and the interaction requirements call for a clear distinction between, on the one hand, *interactive music control* that is associated to a *symbolic time progression* in the underlying *interactive score* and, on the other hand, *music production* that is associated to a *real time clock*.

Last, a basic robustness constraint in such a interactive system is that, during any live performance, the music produced by the system as well as the music produced by any real musician must remain *on time*. More precisely, if ever the computer system (as any musician) turns out to be out of time at some point, then it must auto-synchronize back with the musicians (or any other components) as if nothing had happend. It follows that the system must somehow auto-stabilize w.r.t. the symbolic progression in the score itself.

Main contribution

In this paper, along the research lines exposed in [7], we aim at proposing an abstract generic system architecture for interactive real-time music performance that will fulfill all the above requirements.

Our proposal is based on a lazy real-time kernel that handles, somehow in a reactive way, both scheduled synchronous events and unpredictable asynchronous inputs. The resulting real-time and interactive architecture turns out to be peculiarly robust w.r.t. occasional time drifts. Indeed, whenever forced out of time, the running system auto-stabilizes 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 that have clear and distinct functional specifications. In particular, the symbolic musical control layer can just be seen as an abstract interpreter of any symbolic interactive scores, i.e. our system architecture is generic w.r.t. interactive scores.

Related works and subject position in the field

These last decades have seen the development of many softwares for Computer Assisted Music. This software are increasingly used either on stage for live performances or within multimedia applications that provide thus richer and richer interactive audio supports.

These softwares range from low level sound synthesis and control softwares 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 is to be combined with high level (asynchronous) 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. Most of such systems, see [5] among others as an example, can then just be seen as sorts of *domain specific languages* that are adapted to the design and to the implementation of interactive scores.

Are these languages reaching a good abstraction level ? Do they induce a rich enough notion of interactive scores ? The pragmatic reuses of existing reliable low level softwares somehow mess up the picture. So far, 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.

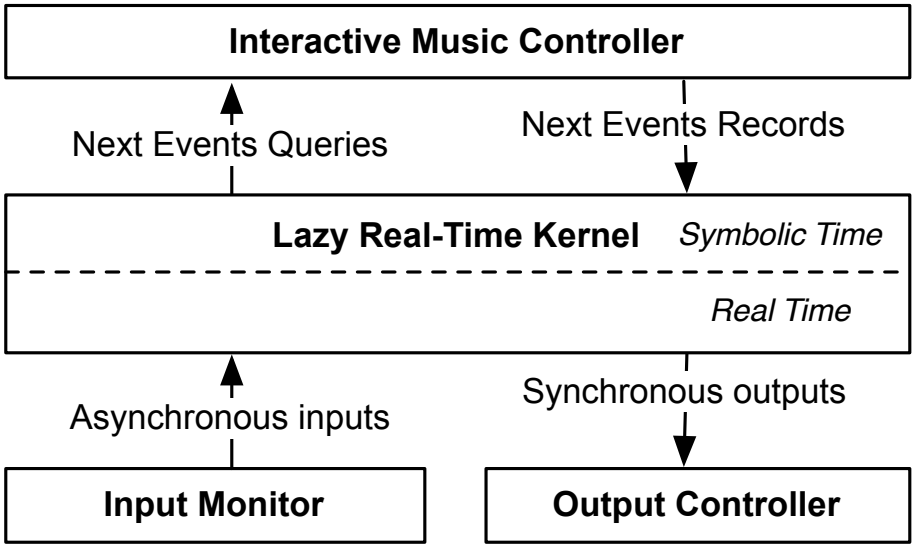
In the present paper, while we do not know *what* exactly is an interactive musical score, we still aim at understanding precisely *where* and *how* such interactive scores can be played.

As a result, we obtain an abstract operational semantics for such scores that

has some similarity with Alur and Dill timed automata [3]. This suggests that some already known models, developed for years in the presumably distinct application context of critical embedded systems, could be adapted efficiently to the application context of interactive music. Even more, some static analysis techniques could be used in order to help composers discovering design errors in their compositions.

2 General architecture

The generic system architecture we propose is essentially composed of three components' layers with components connected as depicted in the following picture.



The functionality of these components is described as follows.

The interactive music controller. This component acts as an interactive partition follower; it receives from the real-time kernel musical events queries with symbolic time stamps and produces back to the real-time kernel the records 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; functionally, it permits 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 itself.

(a) the input subcomponent receives asynchronous control input events from musician instruments and transmit them with no delay to the real-time kernel,

(b) the output subcomponent produces musical output events upon reception of their symbolic descriptions from the real-time kernel.

3 Lazy real time loop

The lazy real time loop, that runs on the real-time kernel, is essentially in charge of the reactive commutation 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.

```
Time * T = M.getNextEventDate();
While (T.isDefined()) do
{
  Event * E = I.WaitEventUntil(T);
  if (E.isDefined())
     M.updateUponReceivedEvent(E,T);
  else
  {
     E = M.getNextEventAtTime(T);
     O.fireEventAtTime(E,T);
  }
  T = M.getNextEventDate();
}
```

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

(1) either an *unpredictable asynchronous event* `E` is received from the input monitor `I` and in that case it is passed with no delay to the music controller that handles it,

(2) or the *scheduled next event date* arrived and the new scheduled synchronous event `E`, available from the music controller `M`, is passed to the output controller to be fired.

The monitoring of these two events is then implemented via the call of the method `WaitEventUntil(T)` on the input monitor `I`. It returns the undefined object `nil` when the actual date is equal or greater than the scheduled date encoded in `T`.

One may also observe that when a new event is received, since it is passed to the music controller, it can be updated, enriched or even just ignore. Indeed, the musical consequence of the reception of an asynchronous event just depends on the interactive score that is followed by the music controller.

For instance, as a particular case, upon reception of an event `E`, a copycat scenario can take place. It is detailled as follows.

Right after being updated upon the reception of event `E`, the next scheduled date computed by the music controller `M` can be defined as the actual reception date of event `E`. It follows that, executing the next the next loop, the `WaitEventUntil(T)` performed by the input monitor `I` immediately returns `nil`. Then the event to be fired computed by `M.getNextEventAtTime(T)` can be defined as the previously received event `E` and it is then just fired by the output controller `O`.

Here and in the remainder of the text, we speak about events, presumably single, but it shall be clear that we rather deals with lists of events that are received or produced at about the same time.

4 Real time vs symbolic time flows

As already mentioned, both input monitor and output controller handle real-time dates while music controller handles symbolic dates. So far, in the lazy real-time loop describe above we have just hidden how these real-time dates are converted into symbolic dates and vice-versa, how these symbolic dates are converted into real-time dates.

The class `Time` that has a single object `T` is in charge of such back and forth conversions. It is described here and in the next section.

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* :

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

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.

We first 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 immediately observe that, provided any change of tempo[1] is modeled as a peculiar event, this hypothesis is just a matter of 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 more general *invariant property* associated with these new attributes:

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

shows, in that 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, controller `M` only handles symbolic time. Converting the real-time date of reception of event `E` into its corresponding symbolic time value is thus a necessity.

5 Scheduled dates and last 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* and *real-time dates* of the next planned events.

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 events,

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

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

These scheduled dates are related with last dates in the same way current dates are related with last dates.

Observe however, that it is the *symbolic scheduled date* that is now first provided by the music controller when computing the next event date. It follows that we need now to compute *real-time scheduled date* from that symbolic value.

The relevant *invariant property* defined by:

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

shows how the value of the *real-time scheduled date* can be computed from the value of the *symbolic scheduled date*.

Observe that 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.

Having said so, we are now ready to describe the update procedure of the *real-time* and the *symbolic last event dates*. 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 that 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 that may be associated to the event fired just occurs right after that update :

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

Observe that, 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.

6 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.

7 Interactive music controller

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 behavior of the interactive music controller.

As described by the lazy real-time loop, at any (symbolic) time, the interactive music controller essentially provides the soonest next scheduled event.

However, before that event is actually fired, any asynchronous external event may occur possibly changing again that soonest next scheduled event.

In other words, this means that the interactive music controller is sort of a timed automaton in the sense of Alur and Dill [3] that can typically read timed input events (the controller is timed reactive) but where, moreover, a default transition is always activated after some fixed elapsed time (the controller is also timed active).

In order to give some more intuition on how such timed controllers 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 beats. At each beat date a metronome click is produced. At any other date, upon reception of an external event `E`, the next beat scheduled date is saved, the next scheduled date is reseted to the current time, the next event to be played at current time is then set to be the received event `E` and is thus produced immediately. After that event has been fired, the next beat scheduled date that has been saved can thus be reactivated as such.

Doing so, an 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.

8 Conclusion

As already mentioned in the text, a prototype of that system has already been implemented in *Objective C* under Mac OSX. It must be added that a more stable implementation is now currently under development in C++ for systems as various as Unix, MacOSX or MS-Windows.

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, which is fairly low level, just seems inadequate for composition. In other words, we still 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 addition, we may consider two complementary research directions for such a development.

The first one concerns the capacity of such a score to describe rather simply musical anticipation. How received events can influence the next events to be produced ? Aside statistical analysis and continuation techniques that are proposed by software such Continuator[15] or OMax[8], we believe that this can also be achieved by a careful encoding of what is known in music as anacrusis [10].

Indeed, any started anacrusis induces some musical coherence constraints that somehow anticipate on the possible musical continuations. This approach already led us to develop a model of overlapping tiles [11] that may well give a rather versatile models of musical patterns.

The second one concerns hierarchization of the description of the music. It seems that composers are in need of ways to think about their music at various abstraction levels.

Hierarchical system modeling techniques are already defined in various areas such as, in particular, statecharts in UML [9]. It seems however that this approach will need to be adapted for hierarchical interactive music descriptions.

More precisely, as suggested in a preliminary study [10], combining musical patterns with overlaps may lead to a rather robust and theoretically well-founded formalism as already illustrated by a rather rich underlying algebraic structure [13, 12].

However, we are still at an exploratory stage. This suggested model still needs to be further developed for an effective use in a composition assistant.

In all case, we expect that the present proposed system architecture, that is easily programable by any notion of interactive scores, 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] 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.

[7] 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.

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

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

[12] David Janin. Quasi-inverse monoids. Technical Report RR-1459-12, LaBRI, Université de Bordeaux, February 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.