Planning and Scheduling Actions in a Computer-Aided Music Composition System

Dimitri Bouche, Jean Bresson

Planning and Scheduling Actions in a Computer-Aided Music Composition System

Dimitri Bouche and **Jean Bresson**
STMS: IRCAM-CNRS-UPMC
1, place Igor Stravinsky, Paris F-75004
{bouche,bresson}@ircam.fr

Abstract

This paper presents a scheduling model for computer music systems. We give an overview of planning and scheduling issues in computer-aided music creation and rendering, and propose strategies for executing actions and computations in music composition or performance contexts.

Introduction

It is well known that the notion of scheduling can imply different levels and complexity in planning tasks and sharing resources (Lawler et al. 1993). Minimizing the resources and optimizing the timing of a process requires a strategy to determine the best ordering of tasks, and every task or computing instruction may itself require a careful planning of operations. In this paper we highlight some specificities of the planning and scheduling processes involved in a computer music application.

Music is a prolific field for computer systems and domain-specific programming environments. Many of them have been developed to support composition and other interactive tasks related to music writing and performance (Dannenberg, Desain, and Honing 1997). Therefore, a variety of applications and computing paradigms exist within computer music environments, implying different perspectives and concerns regarding the notion of scheduling.

We consider a particular subset of computer music systems dedicated to *computer-aided composition* (Assayag 1998). These systems focus on the production and transformation of musical structures, which can be read as scores or rendered by audio players or synthesizers. In computer-aided composition systems the *planning* (generation and ordering of musical actions) and the *execution* (or "rendering") are usually two separate processes which operate sequentially. In this context real-time constraints only concern the execution phase. In the planning phase, musical data can be computed following simple best-effort strategies.

Other types of musical systems are more oriented towards interaction, and process events and audio streams in real-time during music performances (Puckette 1991). In these systems the musical rendering is the output of periodic computations driven by interruptions or callbacks from audio drivers or external systems, which results are produced in bounded and minimal time intervals. Usually in this case, preliminary planning is very basic and complex temporal scenarios can hardly be developed.

Between these two archetypal cases, a number of current projects and software are challenged by the joint management of real-time interaction and the planning of musical structures organised on the longer term (Echeveste et al. 2013; Agostini and Ghisi 2013; Bresson and Giavitto 2014).

In this paper we describe the characteristics and design of a scheduling engine for computer music systems conforming with both compositional applications (i.e. static and independent planning and execution processes) and dynamic/interactive situations (where planning operations occur continually and concurrently with the execution). We introduce a two-fold representation connecting the low-level sequence of actions and the higher-level musical structures involved in score editing and rendering. We successively describe the score planning and scheduling models, and show how they can be made dynamic, allowing planning operations to be part of the execution process.

Score Representations and Planning

The score is a central notion in music composition, considered both as a musical object and as a working environment for composers (see Figure 1). During the process of *rendering*, it is reduced to a sequence of timed actions (notes and other instructions). This process is performed mentally and naturally by musicians interpreting a score, but it has to be carefully designed in an computer rendering system. A planning algorithm (or *planner*) must translate the score into this sequence of actions, by mapping the musical data (pitch, dates etc.) to rendering primitives (functions producing sound from the data).

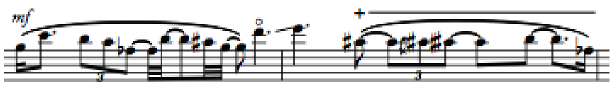


Figure 1: Example of a traditional score.

As contemporary music scores usually include varied kinds of musical data and actions (e.g. sounds, gesture notations, automations for controllers etc. – see Figure 2), the planning strategy must be designed with open and generic representations of both data and actions.

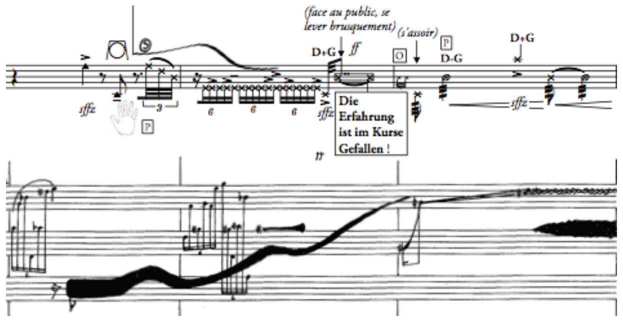

(face au public, se lever brusquement)
(s'assoir)
D+G
D-G
D+G
Die Erfahrung ist im Kurse Gefallen !
sffz
ff

Figure 2: Heterogeneous musical data and controls in a score – extract from *Nachleben* by J. Blondeau (2014).

Planning Model

Let P (the plan) be a list of actions containing rendering operations (e.g. instructions to the audio system, transmission of MIDI[1] messages, or any kind of user-defined actions). Each element in P is an action $a : < t^a, id^a, f^a >$ where:

- t^a is a time-stamp,
- id^a is a unique identifier,
- f^a is a function to execute.

P is a low-level representation optimized for scheduling the rendering process. It must be updated at every modification of the score occurring in the system (for instance from the score editing front-end) and must remain sorted by increasing time-stamps. Three main operations are allowed:

- $schedule(P, a) \equiv$ inserts a at the adequate position in P,
- $unschedule(P, a) \equiv$ removes a from P,
- $reschedule(P, a, t') \equiv$ changes the position of a in P.

Hierarchical Representation

Composers or compositional processes running in a computer-aided composition environment manipulate musical objects with a high degree of structure and hierarchy. A note for instance, which can be considered the minimal specification unit of a musical score, requires at least two distinct actions to be rendered via a MIDI synthesizer: a *key-on*, and a *key-off* action. The *key-on* action must be scheduled at the actual time of the note, and the *key-off* at the time + duration of the note. Between these two actions, *continuous controllers* can also be transmitted to specify the variation of some parameters such as the volume, pitch bending, or other effects implemented in the synthesizer.

One musical object is therefore interpreted as a set of actions. Nevertheless, these actions need to be gathered together in some way in order to ease musical manipulations (a time modification of the note may require the whole set of corresponding actions to be rescheduled). Following the same principle, higher-level musical objects aggregate other objects (*e.g.* a chord gathers several simultaneous notes, a sequence gathers a number of chords under a common time referential) and a hierarchy of musical objects emerges. Hierarchical structures are therefore natural representations for time structures in music (Barbar, Desainte-Catherine, and Miniussi 1993).[2]

The planning model we propose is based on this hierarchical conception, where every musical object has a container and/or a set of children objects. It allows to maintain a correspondence between arbitrarily complex structures manipulated at the musical level and the linear sequence of timed-actions in P.

We consider a musical structure S: $< t^S, id^S, C^S >$ where:

- t^S is a time-stamp relative to the container of S,
- id^S is a unique hierarchical identifier,
- C^S is a list of children objects.

The hierarchical identifiers are constructed by appending a local unique identifier i to the container's id:

$$S_X \in C^{S_Y} \rightarrow id^{S_X} = id^{S_Y}.i$$

Figure 3 shows a graphical representation of a structure S with the following structure:

$$\begin{array}{ll} S & = < 0, 0, [S_1, S_2] > \\ S_1 & = < t_1, 1, [S_{1.1}, S_{1.2}] > \\ S_2 & = < t_2, 2, \emptyset > \\ S_{1.1} & = < t_{1.1}, 1.1, [S_{1.1.1}, S_{1.1.2}, S_{1.1.3}] > \\ S_{1.1.1} & = < 0, 1.1.1, \emptyset > \\ \ldots & \end{array}$$

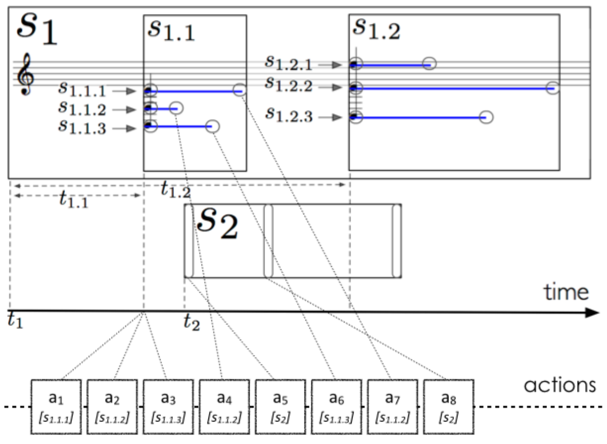


Figure 3: Example of a hierarchical musical structure and conversion to a plan (timed list of actions). S_1 is a sequence structured as a 3-levels hierarchy (sequence/chords/notes). S_2 is a sequence of 3 actions (e.g. parameter changes in a synthesizer's effect controller).

[1]MIDI (Musical Instrument Digital Interface) is a standard protocol and file format for transferring scores and instructions between musical software and digital instruments. MIDI *messages* can be seen as instructions sent to external synthesizers (play/stop note, set volume or effect parameters, etc.)

[2]Similar models have been proposed as well as in other domains, see for instance (Balaban and Murray 1998).

We call A^S is a list of actions a_i directly related to a structure S (not including the ones corresponding to its children). A^S must be specified by the system designer, depending on the type and value of S.

The planner retrieves the set of actions $\overline{A^S}$ corresponding to a hierarchical musical structure S through a recursive traversal of the children tree C^S:

$$P = \overline{A^S} = append(A^S, \bigcup_{S_i \in C^S} \overline{A^{S_i}}) \mid \overline{A^\emptyset} = \emptyset$$

In this process for each $a_j =< t^{a_j}, id^{a_j}, f^{a_j} > \; \in A^{S_i}$:

- The **functions** f^{a_j} depend on the type and value of S_i and are independent from the planning and scheduling model. Their determination is the main task of the system programmer using this model.
- The **time-stamps** t^{a_j} of the actions must be expressed as absolute time values in P. They can be partially derived from the hierarchy of the top-level structure S; for instance, the absolute time of $S_{1.1}$ in Figure 3 is $t^{S_1} + t^{S_{1.1}} = t_1 + t_{1.1}$.[3]
  The specific properties of S_i may also be taken into account to determine t^{a_j} for $a_j \in A^{S_i}$; for instance to a "note" structure N of duration d^N correspond two actions respectively at times t^N and $(t^N + d^N)$.
- The **action identifiers** id^{a_j} are automatically derived from id^{S_i}. For instance, three actions directly related to a structure with identifier $id^{S_i} = a.b$ will be assigned identifiers $id^{a_1} = a.b.1$, $id^{a_2} = a.b.2$ and $id^{a_3} = a.b.3$. This correspondence allows to maintain and retrieve information about the musical structure and hierarchy of S from the planning and scheduling processes.

The basic scheduling operations mentioned previously can then be applied to any musical structure S:

- $schedule(P, S) \equiv schedule(P, a) \; \forall \; a \in \overline{A^S}$
- $unschedule(P, S) \equiv unschedule(P, a) \; \forall \; a \in \overline{A^S}$
- $reschedule(P, S, t') \equiv reschedule(P, a, t'^a) \; \forall \; a \in \overline{A^S}$ with $t'^a = t^a + (t' - t^S)$

Score Rendering and Scheduling

A scheduler must execute the plan P derived from the score (or more exactly, from the musical structure represented in the score), executing all actions $a_i \in P$ on due time. This execution of P can be implemented using standard scheduling and optimization strategies.

Basic Execution model

We note $a_j = P[j]$ the action at position j in P ($j \in \mathbb{N}^+$) and we define a virtual "cursor" position j^P so that $a_{j^P} = P[j^P]$ is the next action in P that the scheduler will execute. At all time t, as P is sorted by increasing t^{a_i}, we will verify that:

$$j^P = min(i \in \mathbb{N}^+ \mid \; t^{a_i} \geq t).$$

The scheduler loop below checks periodically $t^{a_{j^P}}$ against the current clock time and executes actions from the last time interval at each iteration:

Algorithm 1 RENDER(P)

```
loop
    while $t^{a_{j^P}} \leq$ CLOCK_TIME( ) do
        CALL($f^a$)
        $j^P \leftarrow j^P + 1$
    end while
    SLEEP($T$)
end loop
```

Note that past actions (i.e. actions $a_i \mid t^{a_i} < t$ that are already executed at time t) are not removed from P, so that backward modifications and jumps of the cursor remain possible at any time.

With a period T in the order of a millisecond, this simple algorithm will support and render most of the standard musical scores. However, it may be challenged with scores including complex or high-rate sampled data, or if the actions involve computations with execution times that can not be neglected as compared to T. Some strategies for optimizing its execution are discussed further on.

Dealing with Long-term Executions

The system described so far is mostly suitable for dealing with instantaneous actions. In musical systems however (and in particular in the dynamic context we will consider in the next section), we must take into account actions that build or modify musical structures, which might involve arbitrarily complex computations. The execution time can then become a critical point for the correct rendering of the score.

The estimation of this execution time (or of the worst-case execution time – WCET) and its consideration in scheduling systems has been broadly discussed in the literature (Wilhelm et al. 2008). In our case we will consider that either the approximate execution time of an action is known and considered null, or this action falls into the category of "long-term" execution actions. In order to have the renderer loop performing as fast as possible, we will delegate the execution of long-term actions to a separate background process.

The execution strategy adopted for an action a is deduced from f^a: the action is instantly executable if $f^a \in A_{ZT}$, where A_{ZT} is a finite set of functions considered instantaneous in our system ($_{ZT}$ standing for "zero-time"). The macro CALL in Algorithm 1 is therefore defined as follows:

Algorithm 2 CALL(a)

```
if $f^a \in A_{ZT}$ then
    EXECUTE($f^a$)
else
    PROCESS($a$)
end if
```

The call PROCESS(a) in Algorithm 2 sends the execution of the action a outside the scope of the scheduler: the ac-

[3] In the same example, notice for instance that the note's relative time-tags $t^{S_{1.i.j}} = 0$ since these objects are synchronized with their respective containers (the chords $S_{1.1}$ and $S_{1.2}$).

tion is wrapped into a *task* structure and stored in a FIFO queue. The FIFO queue is managed by a *thread-pool* (Kriemann 2004), which dispatches the tasks to *worker* threads: if a worker is available, the task is processed immediately, otherwise it remains in the queue until a worker thread becomes available. Most of the time, the length of the queue remains very small: accumulation happens only when many actions are scheduled in a same, very short time interval.

Tasks are therefore processed as non preemptible in background processes (Henzinger, Horowitz, and Kirsch 2003) and do not impact the timing of the rendering process.

In a musical software, A_{ZT} contains for instance MIDI and external control message senders. In the next sections we will consider examples of more complex actions that are not in A_{ZT}.

Dynamic Planning and Scheduling

In its traditional form, a score is a static structure resulting from a compositional process. It is said static for it does not undergo any modification while being performed or rendered. In computer music systems however it is possible to imagine that an action triggered during the score rendering modifies its own structure (the initial plan). We will speak of an interactive, or *dynamic* score (Desainte-Catherine and Allombert 2005).

Dynamic Scores

In a dynamic score, actions or external events can redefine the plan during its own execution. In this case the scheduling and planning are concurrent processes. The planning is said "continual" (desJardins et al. 1999).

In our model, this dynamic characteristics amounts to allowing the functions f^{a_i} attached to the actions $a_i \in P$ to perform changes on the structure S, and thereby to request updates of P. In other words, actions can invoke the basic scheduling operations *schedule*, *unschedule* and *reschedule*, which respectively schedule new actions, remove and modify previously planned actions. The overall architecture of the system is sketched in Figure 4.

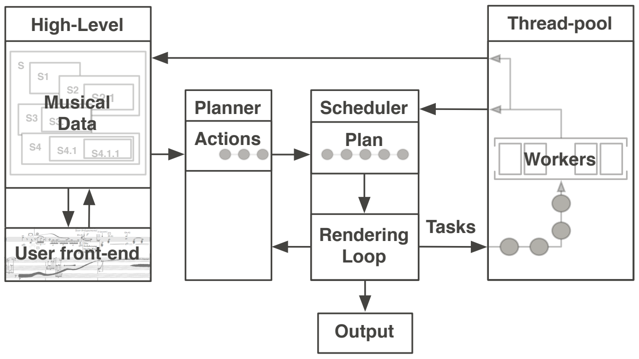


Figure 4: Dynamic architecture. Interactions between high-level structures, planning and scheduling.

The concurrent planning and scheduling operations in the dynamic model require the use of a lock mechanism to secure the concurrent read/write operation on P, as well as efficient sort strategies to be called at adequate moments when the plan is modified.

Extension of the Model

The dynamic score model allows user actions to modify the score S, leading the modification or scheduling of other actions. The result of an action execution if $f^a \notin A_{ZT}$ may therefore affect a musical structure in the middle of its rendering process, and new planning operations may be required immediately when this execution finishes. For this purpose the tasks sent to the *thread pool* are assigned an optional callback returning data to the high-level structure upon completion (see the *Thread-pool* to *High-Level* arrow on Figure 4).

A number of other situations are to be taken into account, such as actions being unscheduled while their associated task is running in the thread pool. This case can be handled if the scheduler stores a pointer to the task in the action structure at transferring it to the thread pool. Scheduling operations on the action can then easily change the state or abort the associated task.

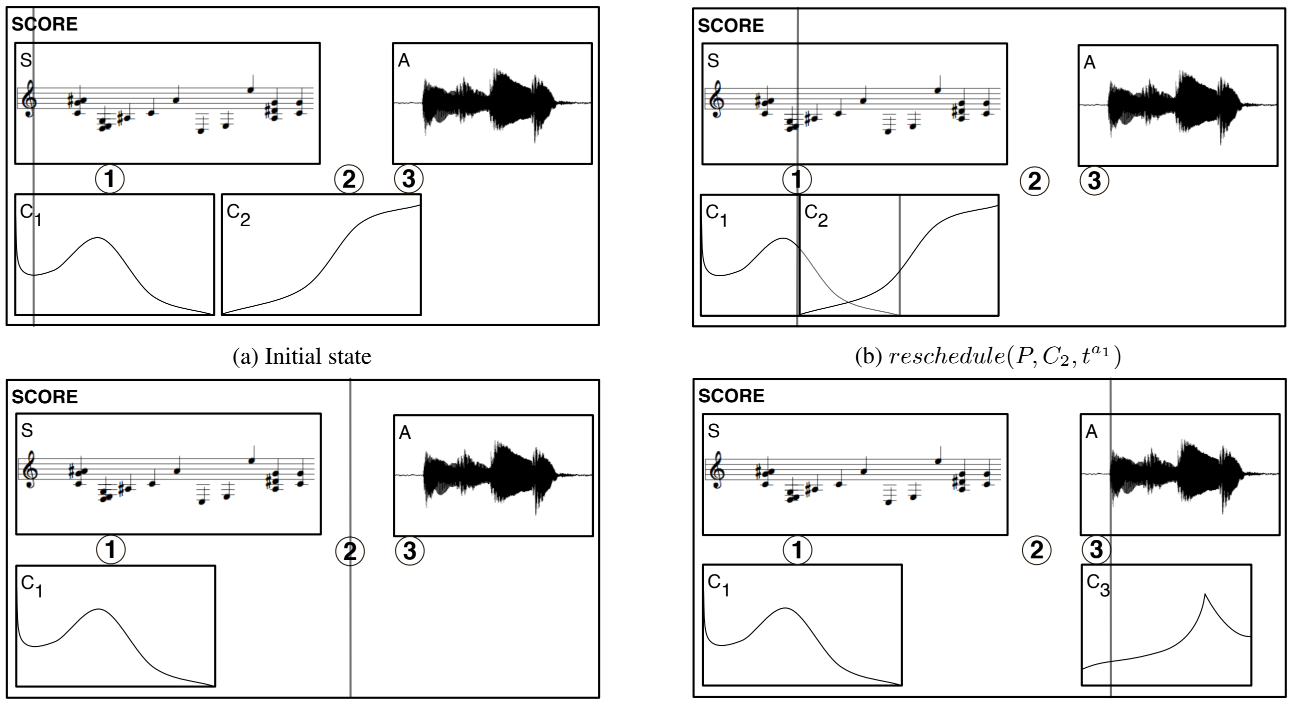
The dynamic model also makes it possible that musical structures be only partially known while the rendering process starts running, which requires considering the availability of the data before to perform actions. It is therefore useful in the score execution to separate functions and data. For this purpose we extend our definition of an action as: $< t^a, id^a, f^a, D^a >$ where D^a is a piece of data used by the function f^a attached to the action (in the general case D^a is a description of the musical structure S). We must then consider the case where the data D^{a_1} required by f^{a_1} and set by f^{a_2} is not available (e.g. if the computation of f^{a_2} does not finish on time). The availability of D^a can be checked by the scheduler prior to the creation of a task for an action a, and behaviours can be determined to react accordingly (e.g. the action a may be skipped, or sent to a thread that will sleep until D^a becomes available). The implementation of such behaviours, though not described here, is done by extending the definition of the action tuple.

Of course this architecture does not guarantee that computations will finish and make any data available on time. However, the score rendering process can run safely delegating actions to the thread pool and reacting to task termination (or non-termination) with predefined behaviours.

Example

In this section we propose a simple score example making use of the dynamic scheduling operations described in the previous sections. The score on Figure 5a contains the following objects:

- A hierarchical sequence of chords and notes (S) rendered as a sequence of MIDI messages,
- An audio file (A) rendered through a standard audio player,
- Two continuous controllers (C_1, C_2) sending values to external audio systems at a high rate (in the order of 100Hz),



(a) Initial state

(b) $reschedule(P, C_2, t^{a_1})$

(c) $unschedule(P, C_2)$

(d) $schedule(P, \text{BUILD}(C_3))$

Figure 5: Score example.

- Special events labelled ①, ② and ③ which respectively:
  - reschedule C_2 to the event's position,
  - unschedule (remove) C_2,
  - build and schedule a new controller (C_3).

We can see this score as a dynamic system controlling sound synthesizers and audio effects. We can imagine for instance that C_1 acts on a parameter of the synthesizer receiving the MIDI notes, and that C_2 and C_3 control audio effects applied to the general audio output. In order to make this dynamic score an interactive one we can also imagine that the events ①, ② and ③ appear dynamically during the execution of the score as the consequences of external events (e.g. performer inputs, sensors, etc.)

The functions f^{a_1}, f^{a_2} and f^{a_3} corresponding to the events ①, ② and ③ can be defined as:

- $f^{a_1} : reschedule(P, C_2, t^{a_1})$
  $\equiv reschedule(P, a_i, t^{a_1} + t^{a_i})$ for each $a_i \in \overline{A^{C_2}}$
- $f^{a_2} : unschedule(P, C_2)$
  $\equiv unschedule(P, a_i)$ for each $a_i \in \overline{A^{C_2}}$
- $f^{a_3} : schedule(P, \text{BUILD}(C_3))$
  $\equiv \text{BUILD}(C_3)$ then $schedule(P, a_i)$ for each $a_i \in \overline{A^{C_3}}$

This score is converted into a plan P by collecting actions from its internal objects. The successive execution states after each event are displayed on Figure 5b, 5c and 5d.

The execution of a score like the one in this example remains continuous despite the dynamic plan modifications. Still, we can notice couple of artefacts in the rendered output. As f^{a_1} moves numerous actions of P, the scheduler can miss the first few rendering actions of C_2 when this object is rescheduled.[4] Similarly, in situations like ③ where an object is computed and immediately scheduled, we can observe a latency between the time a_3 is executed and the time C_3 is effectively scheduled. This latency seems hard to manage due to the unpredictability of the OS-controlled preemptive scheduling environment in which the system runs.

It is important to precise however that both previous remarks are due objects being scheduled on the fly at the exact action times, and would not hold (or would not be detectable) if a reasonable delay is secured between the action times and the newly scheduled objects' dates. Defining operations feasible on time is part of the responsibility of the composer (or of the musical system designer); nevertheless, the estimation and consideration of such delays and constraints in the action planning and execution could be an interesting direction for future works.

Conclusion and Perspectives

The scheduling engine we described implements dynamic features, including the execution of actions with non-deterministic behaviours or execution times, in a musical score renderer system (that is, the kernel of a score-based musical software). The hierarchical structure we propose

[4]Missing events – especially initializing events – can be a serious problem in the control of stateful synthesizers.

permits manipulations at the musical level to be propagated at the low-level of the scheduler, and the scheduler actions to modify the top-level musical representations. At the difference of models such as the Hierarchical Task Network planning (Georgievski and Aiello 2015), the hierarchy here is considered at the level of the user (musical) representations and related planning operations, but remains out of the scope of task executions.

The straightforward approach described in this paper unveils planning and scheduling problematics in computer science applied to music. We are currently comparing it to a number of different approaches, for instance using tree-structured action lists and shorter-term planning.

The system is implemented in the OpenMusic environment (Bresson, Agon, and Assayag 2011). This environment has a wide user base in the contemporary/computer music community, which shall soon provide real-sized situations and use cases to assess its efficiency and reliability.

At a higher, musical level, our future work will concern the interfaces and tools proposed to the musicians that will allow them to take full advantage of the system, for instance for choosing or defining dynamic (re)scheduling actions, or specifying the behaviours of the scheduler regarding the availability of data.

Acknowledgments

This work is funded by the French National Research Agency project with reference ANR-13-JS02-0004. We thank Jérémie Garcia and Jean-Louis Giavitto for their proofreading and numerous suggestions.

References

Agostini, A., and Ghisi, D. 2013. Real-time computer-aided composition with bach. *Contemporary Music Review* 32(1):41–48.

Assayag, G. 1998. Computer Assisted Composition Today. In *1st symposium on music and computers*.

Balaban, M., and Murray, N. 1998. Interleaving Time and Structures. *Computer and Artificial Intelligence* 17(4).

Barbar, K.; Desainte-Catherine, M.; and Miniussi, A. 1993. The Semantics of Musical Hierarchies. *Computer Music Journal* 17(4).

Bresson, J.; Agon, C.; and Assayag, G. 2011. OpenMusic. Visual Programming Environment for Music Composition, Analysis and Research. In *ACM MultiMedia 2011 (OpenSource Software Competition)*.

Bresson, J., and Giavitto, J.-L. 2014. A reactive extension of the openmusic visual programming language. *Journal of Visual Languages and Computing* 4(25):363–375.

Dannenberg, R. B.; Desain, P.; and Honing, H. 1997. Programming Language Design for Music. In Roads, C.; Pope, S. T.; Piccialli, A.; and DePoli, G., eds., *Musical Signal Processing*. Swets and Zeitlinger.

Desainte-Catherine, M., and Allombert, A. 2005. Interactive scores: A model for specifying temporal relations between interactive and static events. *Journal of New Music Research* 34(4):361–374.

desJardins, M. E.; Durfee, E. H.; Charles L. Ortiz, J.; and Wolverton, M. J. 1999. A Survey of Research in Distributed, Continual Planning. *AI Magazine* 20(4).

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

Georgievski, I., and Aiello, M. 2015. Htn planning: Overview, comparison, and beyond. In Elsevier., ed., *Artificial Intelligence*, volume 222, 124–156.

Henzinger, T. A.; Horowitz, B.; and Kirsch, C. M. 2003. Giotto: A time-triggered language for embedded programming. In *Proceedings of the IEEE*, volume 91, 84–99.

Kriemann, R. 2004. *Implementation and Usage of a Thread Pool based on POSIX Threads*. Max-Planck-Institue for Mathematics in the Sciences, Inselstr. 22-26, D-04103 Leipzig, Germany.

Lawler, E. L.; Lenstra, J. K.; Kan, A. H. R.; and Shmoys, D. B. 1993. Sequencing and scheduling: Algorithms and complexity. *Handbooks in OR & MS* 4(445-521).

Puckette, M. 1991. Combining Event and Signal Processing in the Max Graphical Programming Environment. *Computer Music Journal* 15(3).

Wilhelm, R.; Engblom, J.; Ermedahl, A.; Holsti, N.; Thesing, S.; Whalley, D.; Bernat, G.; Ferdinand, C.; Heckmann, R.; Mitra, T.; Mueller, F.; Puaut, I.; Puschner, P.; Staschulat, J.; and Stenström, P. 2008. The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems* 7(3).