MidifilePerformer: a case study for chronologies

Juliette Chabassier, Myriam Desainte-Catherine, Jean Haury, Marin Pobel, Bernard P Serpette

▶ **To cite this version:**

Juliette Chabassier, Myriam Desainte-Catherine, Jean Haury, Marin Pobel, Bernard P Serpette. MidifilePerformer: a case study for chronologies. 26th ACM SIGPLAN International Conference on Functional Programming (ICFP '21), Aug 2021, En ligne, South Korea. pp.13-22, 10.1145/3471872.3472968 . hal-03578817

HAL Id: hal-03578817

https://hal.science/hal-03578817

Submitted on 17 Feb 2022

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.

Abstract

This article discusses the internal architecture of the MidifilePerformer application. This software allows a user to follow a score described in the MIDI format at its own pace and with its own accentuation. MidifilePerformer allows for a wide variety of style and interpretation to be applied to the vast number of MIDI files found on the Internet.

We present here the algorithms enabling the association between the commands made by the performer, via a MIDI or alpha-numeric keyboard, and the notes appearing in the score. We will show that these algorithms define a notion of *expressiveness* which extends the possibilities of interpretation while maintaining the simplicity of the gesture.

MidifilePerformer: A Case Study for Chronologies

Juliette Chabassier, Inria, Bordeaux, France Juliette.Chabassier@inria.fr
Myriam Desainte-Catherine, LaBRI, Bordeaux, France, myriam.desainte-catherine@labri.fr
Jean Haury, Scrime, Bordeaux, France, jeanhaury@gmail.com
Marin Pobel, Université de Bordeaux, Bordeaux, France, marin.pobel@etu.u-bordeaux.fr
Bernard P. Serpette, Inria, Bordeaux, France, Bernard.Serpette@inria.fr

1 Introduction

MidifilePerformer is the heir of the Metapiano [8] which is an instrument, with only a few keys, driving a degraded score where only the order of the notes and their pitches are preserved, the performer is in charge of the intensity and the tempo[1]. The use of the Metapiano makes possible to forget the technical aspects of a performance (large movements of hands and executions of complex chords) to focus on expressivity (*legato*, *staccato*, *rubato*). Since the tempo is provided by the musician, the score does not have to specify time indications and only contains sequences of beginnings and endings of notes linked to the pressing and releasing of keys. Such degraded scores are described in a specific textual format and therefore must be rewritten for each musical piece.

The major specificity of MidifilePerformer is to free itself from the particular format of the Metapiano scores by finding the adequate information in MIDI files[2], and thus gives the possibility to musicians to perform all the files in this format available on the internet. Even if some kinds of information of the Metapiano format are difficult to find in a MIDI file, the later can be considered as a superset of the Metapiano format. The effort is therefore in the simplification of the score unlike related works, such as Antescofo [5], which allows complex annotations linking the score and the performance.

The implementation of the MidifilePerformer is based on a *model-command-render* triplet. A musician reads a score, operates an instrument and produces music. This same analogy can be used in many fields, to assemble a piece of furniture delivered in a kit, to make a dish etc. In all cases, we identify three entities. The model is the score, the instructions for use, the cooking recipe. The commands represent the actions to be taken to follow the model, play an instrument, tinker, cook. The rendering is the result in progress, the performance, the furniture, the cooking dish. Note that, unlike the model-view-controller[2], which forms a cycle where the user is inserted, the two entities that are the model and the commands are independent[3], the rendering is here to *synchronize* these two entities. Even if it is natural that a human initiates the commands, this is not a strong condition for the MidifilePerformer since some commands can be generated by the computer. .

MidifilePerformer fits into this three-part concept. The model is a score in the form of a MIDI file. The commands come, in real time, from computer equipment, MIDI or computer keyboard. The rendering produces sound, according to the rhythm and the velocity of the commands, and the pitches of the model. For a computer keyboard, a specific velocity is assigned to each character. Any device that can provide, in real time, a velocity can be used as input for commands.

We see that the three concepts integrate different facets of the notion of time. The model is *out of time*, it has no present, but defines stages in time. Commands are anchored in the present, can have a knowledge of what was already done, can have a notion of what remains to be performed, but speak in

[1] Two Metapiano interpretations can be listen on the following addresses: https://www.youtube.com/watch?v=OhDlWxA6SlY and https://www.youtube.com/watch?v=Ul8dfY72TWQ

[2] https://en.wikipedia.org/wiki/Model-view-controller

[3] the triplet could be renamed command-model-render

the present tense: I do that. The rendering is also anchored in the present but is only the result of the commands. Thus, what we are trying to formalise is a function that produces the rendering, according to the model and the commands. We will call this function the render function.

Each of these three entities, model, command and render, is the agglomeration of simpler elements that are events: a thing, whether it is a note, an action or a sound, associated with a time. As time is a unit that can be ordered (past, present, future), it is natural to agglomerate events with increasing time. We read things in the order they are supposed to happen. It is for this reason that we will call these sets of events *chronologies.*

The heart of the MidifilePerformer process is therefore to analyse the last command carried out in relation to what has already been carried out in the model and to produce a result in accordance with the command and the model.

First, we will define more formally the concepts of events and chronology. Then, we will analyse various possibilities of combining a command with a model. Finally, we will give some hints of the implementation.

2 Events and Chronologies

We saw in the introduction that time plays an important role in the model-command-render triplet. This role is even more important in the musical field. Nevertheless, time has little value in itself, it must necessarily be attached to something, a structure, an action, the parameter of a function etc. In general we will call *object* any timeless thing, the number 10 for example. The association of an object and a time forms the *event.* The temperature of 10 degrees is an object, it was 10 degrees this morning is an event. We also speak of temporal objects.

A decisive choice is to specify the mathematical domain of time. Either we associate it with real numbers and we speak then of continuous time, or we identify it with natural numbers, in this case we speak of discrete time. The choice of rational numbers can be interesting, this set is countable, so there are *as many* rational numbers as integers, yet it is dense, between any two rational numbers we can always find a third one between them. But, in the computer field, things are simplified since we usually use finite domains, integers or floats. The difference between the two is a matter of density, for floats there are as many objects between 0 and 1 as between 1 and the largest floating number. While for integers the objects are uniformly distributed. It is this property which pushed us to use a discrete time, we took the milli-second as unit of time. We will assume that $Time$ denotes the set of time values.

So, if T is the type of an object class, then the cartesian product $T \times Time$ denotes the set of events on T.

$$Event(T) \triangleq T \times Time$$

If an event corresponds to an object m occurring at time t, we will denote it m^t.

In many examples, time is spread over an interval, *it rained from noon until 2am.* In this case, it is practical to make a separation between the beginning and the end of the object that we are handling (the rain in our example). If m is such an object occurring over an interval (t_1, t_2), we will denote $\underline{m}^{t_1}$ its beginning and $\overline{m}^{t_2}$ its end. Nevertheless, this object can undergo variations in the interval considered, the rain will not have the same intensity in its duration. Concretely, when using a MIDI object, $\underline{m}$ corresponds to the start of a note (`NOTE_ON`) and $\overline{m}$ to the end of this note (`NOTE_OFF`). When an object m has neither start nor end, this corresponds to a punctual event happening at the instant t, we will denote it $\dot{m}^t$. Continuous events, such as provided by a pitch wheel for example, can be seen as series of punctual events. The algorithms described in this article consider mainly events with a start and an end and therefore are related to Allen's interval algebra[1] .

In this article, we will focus on sets of events. We will call these sets *chronologies* since we will rely heavily on the fact that time is totally ordered.

$$\mathcal{C}(T) \triangleq Chronology(T) \triangleq \mathcal{P}(Event(T))$$

In terms of notation, rather than using the usual writing of sets $\{m_1^{t_1}, m_2^{t_2} \ldots\}$, the chronologies will be described in the form of words $m_1^{t_1} m_2^{t_2} \ldots$, where, implicitly, $t_1 \leq t_2 \leq \ldots$ The special case where $t_1 = t_2$

will be handled by some specific analysis that we will show later.

The definition we have given for chronologies seems static, all the elements of a chronology are known in advance. This is the case for a midi file. On the other hand, the commands are interactive: they are potentially infinite words whose first event is only known when it happens. First, we will expose the algorithms with static chronologies, but taking care to extract the elements in increasing order of time. This precaution will allow us, in the implementation, to consider the chronologies as flows.

3 Functional Definition of MidifilePerformer

We are going to expose the MidifilePerformer engine here without going into the details of the chronology implementations, that will be exposed in the next section.

3.1 Rendering Functions

The MidifilePerformer is centered around a rendering function that combines a model and commands to produce an output called the render. To be independent of *MIDI* objects that will be used at user level, we have developed a low-level layer with generic types, ie types of which we do not know the implementation, but of which we can have a partial view of their properties. For example, for certain treatments, we will impose that a command can be determined as being a start or an end of an interval, or neither.

Thus, if we abstract the elements of the model by a type T_1, the commands by a type T_2, a rendering function is a function $\mathcal{C}(T_1), \mathcal{C}(T_2) \rightarrow \mathcal{C}(T_3)$ where T_3 represents the elements of the render. Of course, this function combines objects of type T_1 with those of type T_2 to produce objects of type T_3, but it will leave the choice of the implementation of this combination at the time of the concrete definition of the abstract types. On the other hand, the main role of the rendering function will be to make *synchronization* between the model and the commands, it must *associate*, in time, events coming from one side with those coming the other.

Time is a concrete value of the elements of a chronology. A rendering function can therefore rely on these time values to make decisions. These will be developed according to general principles that we will discuss.

3.2 Principles

The rendering functions are very general, however some fundamentals have guided those that have been implemented for the MidifilePerformer. We give here principles based on musical concepts, they can be adopted on other categories.

1. The order of the beginnings of the notes in the score must be respected. It is a strong constraint, but natural in our framework, which makes it possible to restrict the algorithms that we will study. On the other hand, this constraint associated with the beginnings of notes could be relaxed for their ends. It may be that a render function decides that an end of a note occurs after the beginning of another when the score specified otherwise.

2. We prevent the release of a command from generating the start of a note. It is a counterintuitive phenomenon. On the other hand, pressing a command can cause the end of a note if at the same time a new note will be started.

3. We avoid as much as possible the inoperative commands. This must necessarily be the case for the enfoncement of a command. Intuitively, one would be surprised that nothing happens when a mechanism is triggered. Nevertheless, it will be tolerated that the release of a command has no effect. For example, it is natural that, under the effect of a pedal, the release of a piano key does not muffle the played note.

These principles have conditioned the algorithms of rendering functions that we present now.

3.3 Algorithms

3.3.1 Model Chronology Processing

As a first step, it is necessary to concretize the notion of simultaneity by grouping together the events having the same time. For example the chronology of the model:

$$\underline{m_1}^{t_1}\underline{m_2}^{t_1}\overline{m_1}^{t_2}\underline{m_3}^{t_2}\dot{m_4}^{t_3}\overline{m_2}^{t_4}\overline{m_3}^{t_4}$$

turns into:

$$\{\underline{m_1},\underline{m_2}\}^{t_1}\{\overline{m_1},\underline{m_3}\}^{t_2}\{\dot{m}_4\}^{t_3}\{\overline{m_2},\overline{m_3}\}^{t_4}$$

Thus the beginnings of events m_1 and m_2 are grouped together at time t_1, in the same way the two endings of events m_2 and m_3 are grouped together at time t_4. We notice, as for the time t_2, that the beginnings and the ends of events (different) can be grouped together. If the elements of the model are of type T_1, this stage of preprocessing of the model has the signature $\mathcal{C}(T_1) \rightarrow \mathcal{C}(\mathcal{P}(T_1))$. We will denote by M_i the elements of $\mathcal{P}(T_1)$ and we will take the notational convention $\underline{M}$ for sets with at least one starting note and $\overline{M}$ for all other cases. So the previous example is rewritten in:

$$\underline{M_1}^{t_1}\underline{M_2}^{t_2}\overline{M_3}^{t_3}\overline{M_4}^{t_4}$$

with $M_1 = \{\underline{m_1},\underline{m_2}\}$, $M_2 = \{\overline{m_1},\underline{m_3}\}$, $M_3 = \{\dot{m}_4\}$ and $M_4 = \{\overline{m_2},\overline{m_3}\}$.

We can consider this preprocessing of the model as a creation of an S-word [6].

The next step consists in constructing an alternation of $\underline{M}$ and $\overline{M}$, even if it means introducing empty sets and merging sets that do not have a starting event (the $\overline{M}$). For example, the previous sequence is rewritten:

$$\underline{M_1}^{t_1}\overline{M'_1}^{t'_1}\underline{M_2}^{t_2}\overline{M'_2}^{t'_2}$$

with $M'_1 = \emptyset$, et $M'_2 = \{\dot{m}_4,\overline{m_2},\overline{m_3}\}$.

As it is the events of the commands that will define the times of the results, the values of t'_1 and t'_2 are not predominant, in the current implementation $t'_1 = t_2 - 1$, just before the following start, and $t'_2 = t_4$, the maximum of the merged times.

An optional step is to avoid empty sets in the following cases: $\underline{m_j} \in \underline{M_i}$, $\overline{M'_i} = \emptyset$ and $\overline{m_j} \in \underline{M_{i+1}}$, which is verified for $i = j = 1$ in the previous example. In this case, the end of note $\overline{m_j}$ is shifted ahead in time in order to take the empty place of $\overline{M'_i}$. Thus, with this option considered, the result would be: $M_1 = \{\underline{m_1},\underline{m_2}\}$, $M'_1 = \{\overline{m_1}\}$, $M_2 = \{\underline{m_3}\}$ and $M'_2 = \{\dot{m}_4,\overline{m_2},\overline{m_3}\}$. We observe that, without this option, the first key release coming from the controls will not produce any effect, whereas with this option, this release has the opportunity to control the end of note $\overline{m_1}$.

3.3.2 Merging Model and Commands

Once the transformations have been applied to the model to obtain a chronology in the form $\ldots\underline{M_i}^{x_i}\overline{M'_i}^{x'_i}\ldots$, the ideal situation for the render function would be that the command chronology is also in the form:

$$\ldots\underline{c_i}^{t_i}\overline{c'_i}^{t'_i}\ldots$$

That is to say an alternation of depression and relaxation. In that case. the render function would merge the events from the two chronologies, one by one, to produce:

$$\ldots(\underline{M_i}\oplus\underline{c_i})^{t_i}(\overline{M'_i}\oplus\overline{c'_i})^{t'_i}\ldots$$

We notice that the times are taken in the commands. The $\oplus$ merge operator will build a set of objects of type T_3 from a set of objects of type T_1 coming from the model and an object of type T_2 coming from the commands . This operator can be defined from a simpler operator $\odot$ of type $T_1 \times T_2 \rightarrow T_3$:

$$M \oplus c \stackrel{\triangle}{=} \{m \odot c / m \in M\}$$

The operator $\odot$ is obviously dependent on the three types considered, but in the context of musical objects used by MidifilePerformer, the operator $\odot$ takes the velocity (in other words the gain or the volume) in the command and the pitch (frequency) in the model.

Of course, this optimistic vision, where the controls are well arranged by alternating pressing and releasing, is not necessarily achieved. Commands can generate all kinds of overlaps like:

$$\underline{c_0}^{t_0}\underline{c_1}^{t_1}\overline{c_1}^{t_2}\overline{c_0}^{t_3}\ldots$$

In Allen's interval algebra terminology [1], in this particular case, we observe here a *during*, the end of the second note appears before the end of the first. If we want to merge these commands with a model $\underline{M_0}\overline{M'_0}\underline{M_1}\overline{M'_1}$, we can consider that, in all cases, in order to associate the start of intervals, the events $(\underline{M_0} \oplus \underline{c_0})^{t_0}$ and $(\underline{M_1} \oplus \underline{c_1})^{t_1}$ have to be generated. For the rest, i.e. the end of intervals, we can consider 3 cases:

1. we respect the temporal order of the model. We generate $(\overline{M'_0} \oplus \overline{c_0})$, in an artificial way, at time t_1. We therefore produce $(\underline{M_0} \oplus \underline{c_0})^{t_0}$ $(\overline{M'_0} \oplus \overline{c_0})^{t_1}$ $(\underline{M_1} \oplus \underline{c_1})^{t_1}$ $(\overline{M'_1} \oplus \overline{c_1})^{t_2}$. Note that at time t_1, we do not necessarily have access to all the information of the end of the interval $\overline{c_0}$ which will be produced at time t_3. It will therefore be necessary to take default values. In general, in the context of MIDI files, the end of notes do not contain any information other than pitch.

2. one respects the order of the end of interval of the model. It is imperative that the effect produced by $\overline{M'_0}$ be generated before the effect produced by $\overline{M'_1}$, we therefore generate $(\overline{M'_0} \oplus \overline{c_1})$ at time t_2 and $(\overline{M'_1} \oplus \overline{c_0})$ at time t_3. We therefore produce $(\underline{M_0} \oplus \underline{c_0})^{t_0}$ $(\underline{M_1} \oplus \underline{c_1})^{t_1}$ $(\overline{M'_0} \oplus \overline{c_1})^{t_2}$ $(\overline{M'_1} \oplus \overline{c_0})^{t_3}$.

3. we respect the association of start and end of interval. We generate $(\overline{M'_1} \oplus \overline{c_1})$ at time t_2 and $(\overline{M'_0} \oplus \overline{c_0})$ at time t_3. We therefore produce $(\underline{M_0} \oplus \underline{c_0})^{t_0}$ $(\underline{M_1} \oplus \underline{c_1})^{t_1}$ $(\overline{M'_1} \oplus \overline{c_1})^{t_2}$ $(\overline{M'_0} \oplus \overline{c_0})^{t_3}$.

These three cases are described more formally in Figures 1 and 2 by the functions $\texttt{combine}_1$, $\texttt{combine}_2$ and $\texttt{combine}_3$. We have also introduced a new case via the function $\texttt{combine}_0$ which we will explain later.

These functions have a similar profile. They all take as parameter the chronology μ of the model having undergone the processing described previously, the chronology σ of the commands and potentially some auxiliary structures, π and τ, depending on the cases considered. These functions make a case study according to the chronologies of model and commands. The first case is when we can find in the model a series of two elements $\underline{M_1}$ and $\overline{M_2}$ and the commands begin with an interval start $\underline{c}^t$. In this case, the event $(\underline{M_1} \oplus \underline{c})^t$ will have to be emitted. It will also be necessary to memoize the events contained in $\overline{M_2}$ in order to restore them later, this is why the variable π intervenes. The second case is when the commands begin with an end of interval. In this case, the events to be sent must have been stored in the π structure. The third case concerns punctual events. Regardless of the case, it is necessary to consider, via the function `filter`, whether this event must be returned by the rendering function. The last case corresponds to the end of one of the two chronologies.

The $\texttt{combine}_0$ function is special. It considers that the orders cannot include any end of interval. This is the case, for example, for one of the game modes of bao-pao or *metaclaquettes*[4] where commands are always punctual events. These end of intervals, $\overline{M_2} \oplus \overline{c}$, are saved in the variable π each time a command is received, and emitted, in the form π^t, upon receipt of the next command. The τ variable is used to store the time of the last command, in order to issue the last end of the interval.

The function $\texttt{combine}_1$ is similar to $\texttt{combine}_0$ except that a releasing command $(\overline{c}^t)$ can trigger an end of interval $(\overline{M_2} \oplus \overline{c})$, if it occurs just after the related depression command. Otherwise, the behaviour will be the same as that of the function $\texttt{combine}_0$. For a model of the form $\underline{M_1}\overline{M'_1}\underline{M_2}\overline{M'_2}$, if the commands are in the form $\underline{c_1}^{t_1}$ $\overline{c_1}^{t'_1}$ $\underline{c_2}^{t_2}$ $\overline{c_2}^{t'_2}$ $\ldots$, then, unsurprisingly, the rendering will be $(\underline{M_1} \oplus \underline{c_1})^{t_1}$ $(\overline{M'_1} \oplus \overline{c_1})^{t'_1}$

[4] https://www.bao-pao.com

$\texttt{combine}_0(\mu,\sigma,\pi,\tau) \triangleq$
  match (μ,σ) **with**
  $\mid (\underline{M_1}^x\cdot\overline{M_2}^y\cdot\mu'), (\underline{c}^t\cdot\sigma') \rightarrow$
    $\pi^t\cdot(\underline{M_1}\oplus\underline{c})^t\cdot\texttt{combine}_0(\mu',\sigma',\overline{M_2}\oplus\overline{c},t)$
  $\mid _, (\dot{c}^t\cdot\sigma') \rightarrow$
    $\texttt{filter}(\dot{c},t)\cdot\texttt{combine}_0(\mu,\sigma',\pi,t)$
  $\mid _, _ \rightarrow \pi^\tau$

$\texttt{combine}_1(\mu,\sigma,\pi,\tau) \triangleq$
  match (μ,σ) **with**
  $\mid (\underline{M_1}^x\cdot\overline{M_2}^y\cdot\mu'), (\underline{c}^t\cdot\sigma') \rightarrow$
    $\pi^t\cdot(\underline{M_1}\oplus\underline{c})^t\cdot\texttt{combine}_1(\mu',\sigma',\overline{M_2}\oplus\overline{c},t)$
  $\mid _, (\overline{c}^t\cdot\sigma') \rightarrow$
    $\pi^t\cdot\texttt{combine}_1(\mu,\sigma',\emptyset,t)$
  $\mid _, (\dot{c}^t\cdot\sigma') \rightarrow$
    $\texttt{filter}(\dot{c},t)\cdot\texttt{combine}_1(\mu,\sigma',\pi,t)$
  $\mid _, _ \rightarrow \pi^\tau$

Figure 1: combine$_0$ and combine$_1$

$\texttt{combine}_2(\mu,\sigma,\pi) \triangleq$
  match (μ,σ,π) **with**
  $\mid (\underline{M_1}^x\cdot\overline{M_2}^y\cdot\mu'), (\underline{c}^t\cdot\sigma') \rightarrow$
    $(\underline{M_1}\oplus\underline{c})^t\cdot\texttt{combine}_2(\mu',\sigma',\overline{M_2}\cdot\pi)$
  $\mid _, (\overline{c}^t\cdot\sigma'), \pi'\cdot\overline{M} \rightarrow$
    $(\overline{M}\oplus\overline{c})^t\cdot\texttt{combine}_2(\mu,\sigma',\pi')$
  $\mid _, (\dot{c}^t\cdot\sigma') \rightarrow$
    $\texttt{filter}(\dot{c},t)\cdot\texttt{combine}_2(\mu,\sigma',\pi)$
  $\mid _, _, _ \rightarrow []$

$\texttt{combine}_3(\mu,\sigma,\pi) \triangleq$
  match (μ,σ,π) **with**
  $\mid (\underline{M_1}^x\cdot\overline{M_2}^y\cdot\mu'), (\underline{c}^t\cdot\sigma') \rightarrow$
    $(\underline{M_1}\oplus\underline{c})^t\cdot\texttt{combine}_3(\mu',\sigma',(\overline{c},\overline{M_2})\cdot\pi)$
  $\mid _, (\overline{c}\cdot{}^t\sigma'), \pi_1\cdot(\overline{c},\overline{M})\cdot\pi_2 \rightarrow$
    $(\overline{M}\oplus\overline{c})^t\cdot\texttt{combine}_3(\mu,\sigma',\pi_1\cdot\pi_2)$
  $\mid _, (\dot{c}^t\cdot\sigma') \rightarrow$
    $\texttt{filter}(\dot{c},t)\cdot\texttt{combine}_3(\mu,\sigma',\pi)$
  $\mid _, _, _ \rightarrow []$

Figure 2: combine$_2$ and combine$_3$

$(\underline{M_2}\oplus\underline{c_2})^{t_2}$ $(\overline{M_2'}\oplus\overline{c_2})^{t_2'}$. On the other hand, for commands $\underline{c_1}^{t_1}$ $\underline{c_2}^{t_2}$ $\overline{c_1}^{t_1'}$ $\overline{c_2}^{t_2'}$ ..., then the rendering becomes: $(\underline{M_1}\oplus\underline{c_1})^{t_1}$ $(\overline{M_1'}\oplus\underline{c_1})^{t_2}$ $(\underline{M_2}\oplus\underline{c_2})^{t_2}$ $(\overline{M_2'}\oplus\overline{c_2})^{t_2'}$. In this case, the release command $\overline{c_1}$ done at time t_1' has no effect on the output, the events $(\overline{M_1'}\oplus\underline{c_1})$ are generated only with informations provided by $\underline{c_1}$, moreover these events are emitted at time t_2 and nothing will be done at time t_1'.

The function $\texttt{combine}_2$ keeps the end of intervals $\overline{M_2}$ in a list stored in π. These end of intervals are inserted from the left $(\overline{M_2}\cdot\pi)$ and extracted from the right $(\pi'\cdot\overline{M})$ thus denoting a queue structure (FIFO). A similar version using a stack structure (LIFO) is immediate. We will analyse this version under the name *case 2'*. With the same two examples seen for the $\texttt{combine}_1$ function, we get the same result for the first case, on the other hand, for the second we get: $(\underline{M_1}\oplus\underline{c_1})^{t_1}$ $(\underline{M_2}\oplus\underline{c_2})^{t_2}$ $(\overline{M_1'}\oplus\overline{c_1})^{t_1'}$ $(\overline{M_2'}\oplus\overline{c_2})^{t_2'}$, which allows us to regain control of the time t_1' that we had lost with $\texttt{combine}\ _1$. For this same second example, considering *case 2'*, by inverting the endings of the model interval, we obtain: $(\underline{M_1}\oplus\underline{c_1})^{t_1}$ $(\underline{M_2}\oplus\underline{c_2})^{t_2}$ $(\overline{M_2'}\oplus\overline{c_1})^{t_1'}$ $(\overline{M_1'}\oplus\overline{c_2})^{t_2'}$, which transforms, from a musical point of view, a desire for monophonic *overlap* from the performer, into a rendering of polyphonic *during* . Conversely, a will of *during* turns into *overlap* with *case 2*, while it is respected in *case 2'*.

The function $\texttt{combine}_3$ keeps the end of intervals $\overline{M_2}$ in a list stored in π, taking care to associate them with the command that triggered the start of the interval $(\overline{c})$. π is therefore an association list from which we extract the end of the interval of the model corresponding to the command. Thus the two drawbacks, which were observed for the previous case, disappear.

3.3.3 Expressiveness

To analyse more precisely the differences between the various algorithms, we will compare their behaviour with respect to the set of possible model configurations containing two notes and reacting to a combination of two commands.

Allen's interval algebra is a good support for studying the relative positions of two notes in time and

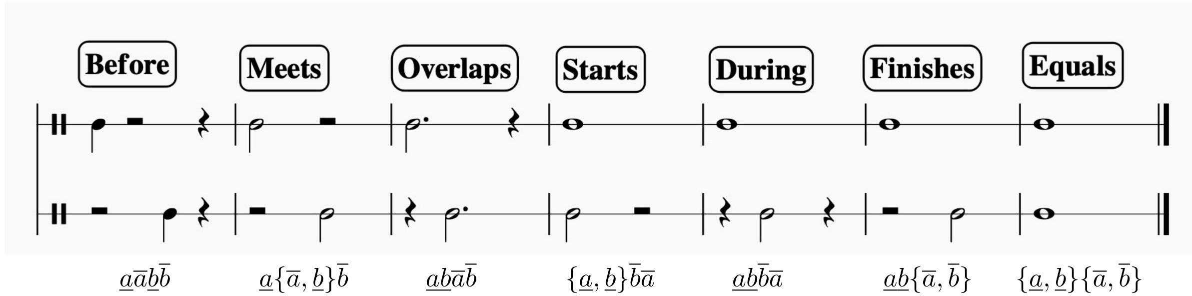

Figure 3: Allen's 7 Different Time Configurations

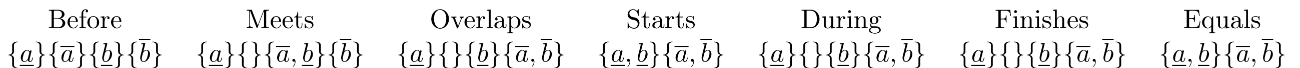

Figure 4: configurations after pre-treatment

in duration. Indeed, the figure 3 shows the seven arrangement configurations of two notes. We give both the names coming from Allen's algebra, their musical notations and their symbolic descriptions where "a" denotes the note on the top and "b" that of the bottom, the set notation is used to specify the simultaneity of the two events. Of Allen's thirteen possibilities, we have gone to seven by omitting the symmetrical versions which we obtain by inverting the two staves: we have arbitrarily chosen that the note of the upper staff begins before that of the lower.

Moreover, in a score, and therefore in the chronology of the model, all these configurations are possible, common even, as some are inaccessible for the commands. Indeed, the MIDI media do not integrate the notion of simultaneity: if a two-note chord is placed on a MIDI keyboard, these two notes will always be serialized, even if it means separating them by a minimum duration (one milli-second). If we take the seven configurations of figure 3, only three do not include simultaneous events (either at the beginning or at the end): Before, Overlaps and During.

Using these same seven configurations, if we perform the processing specified in 3.3.1, we obtain the overall chronologies of figure 4.

Note that the three configurations *Overlaps*, *During* and *Finishes* provide the same overall chronology: from the moment it was decided that $\overline{a}$ and $\overline{b}$ had to be generated from the same release command, we lose the information of the temporal positioning of $\overline{a}$ with respect to $\overline{b}$ in the model. In the same way, we no longer differentiate a *Starts* from a *Equals*. So, on the model side, we will only consider the *Before*, *Meets*, *Finishes* and *Equals* configurations, and on the commands side, the *Before*, *Overlaps* and *During* configurations.

We can compare the various algorithms by means of a matrix where these algorithms are displayed in rows with the various possibilities of two command intervals, and, in columns, the possible configurations of two model intervals. At the intersection of a row and a column we will find the result of the algorithm applied to the two chronologies of commands and models. This matrix is given in the figure 5. We do not mention the case of *Equals* ($\{\underline{a},\underline{b}\}\{\overline{a},\overline{b}\}$) because this configuration involves only one interval in the chronology.

For example, let us take the case 2 corresponding to the use of the function $\texttt{combine}_2$ of the figure 2, consider the second line of this case where the commands are in the form $\underline{x}^0\underline{y}^1\overline{x}^2\overline{y}^3$, which forms an *Overlaps* in Allen's intervals or a *legato* in the musical framework; suppose the model describes a *Meets*, so a chronology of the form $\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^2$ which, after processing, changes to the overall chronology $\{\underline{a}\}\{\}\{\overline{a},\underline{b}\ \}\{\overline{b}\}$. We do not mention the times in the elements of the model because these will be ignored by the algorithms by systematically taking those coming from the commands. For this example, the successive steps of the algorithm are described in the following figure:

		Before $\{\underline{a}\}\{\overline{a}\}\{\underline{b}\}\{\bar{b}\}$	Meets $\{\underline{a}\}\{\}\{\overline{a},\underline{b}\}\{\bar{b}\}$	Overlaps/During/Finishes $\{\underline{a}\}\{\}\{\underline{b}\}\{\overline{a},\bar{b}\}$
case 0	$\underline{x}^0\underline{y}^1$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^2$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^2$	$\underline{a}^0\underline{b}^1\overline{a}^2\overline{b}^2$
case 1	$\underline{x}^0\overline{x}^1\underline{y}^2\overline{y}^3$	$\underline{a}^0\overline{a}^1\underline{b}^2\overline{b}^3$	$\underline{a}^0\overline{a}^2\underline{b}^2\overline{b}^3$	$\underline{a}^0\underline{b}^2\overline{a}^3\overline{b}^3$
	$\underline{x}^0\underline{y}^1\overline{x}^2\overline{y}^3$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^2$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^2$	$\underline{a}^0\underline{b}^1\overline{a}^2\overline{b}^2$
	$\underline{x}^0\underline{y}^1\overline{y}^2\overline{x}^3$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^2$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^2$	$\underline{a}^0\underline{b}^1\overline{a}^2\overline{b}^2$
case 2	$\underline{x}^0\overline{x}^1\underline{y}^2\overline{y}^3$	$\underline{a}^0\overline{a}^1\underline{b}^2\overline{b}^3$	$\underline{a}^0\overline{a}^2\underline{b}^2\overline{b}^3$	$\underline{a}^0\underline{b}^2\overline{a}^3\overline{b}^3$
	$\underline{x}^0\underline{y}^1\overline{x}^2\overline{y}^3$	$\underline{a}^0\underline{b}^1\overline{a}^2\overline{b}^3$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^3$	$\underline{a}^0\underline{b}^1\overline{a}^3\overline{b}^3$
	$\underline{x}^0\underline{y}^1\overline{y}^2\overline{x}^3$	$\underline{a}^0\underline{b}^1\overline{a}^2\overline{b}^3$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^3$	$\underline{a}^0\underline{b}^1\overline{a}^3\overline{b}^3$
case 2'	$\underline{x}^0\overline{x}^1\underline{y}^2\overline{y}^3$	$\underline{a}^0\overline{a}^1\underline{b}^2\overline{b}^3$	$\underline{a}^0\overline{a}^2\underline{b}^2\overline{b}^3$	$\underline{a}^0\underline{b}^2\overline{a}^3\overline{b}^3$
	$\underline{x}^0\underline{y}^1\overline{x}^2\overline{y}^3$	$\underline{a}^0\underline{b}^1\overline{b}^2\overline{a}^3$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^2$	$\underline{a}^0\underline{b}^1\overline{a}^2\overline{b}^2$
	$\underline{x}^0\underline{y}^1\overline{y}^2\overline{x}^3$	$\underline{a}^0\underline{b}^1\overline{b}^2\overline{a}^3$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^2$	$\underline{a}^0\underline{b}^1\overline{a}^2\overline{b}^2$
case 3	$\underline{x}^0\overline{x}^1\underline{y}^2\overline{y}^3$	$\underline{a}^0\overline{a}^1\underline{b}^2\overline{b}^3$	$\underline{a}^0\overline{a}^2\underline{b}^2\overline{b}^3$	$\underline{a}^0\underline{b}^2\overline{a}^3\overline{b}^3$
	$\underline{x}^0\underline{y}^1\overline{x}^2\overline{y}^3$	$\underline{a}^0\underline{b}^1\overline{a}^2\overline{b}^3$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^3$	$\underline{a}^0\underline{b}^1\overline{a}^3\overline{b}^3$
	$\underline{x}^0\underline{y}^1\overline{y}^2\overline{x}^3$	$\underline{a}^0\underline{b}^1\overline{b}^2\overline{a}^3$	$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^2$	$\underline{a}^0\underline{b}^1\overline{a}^2\overline{b}^2$

Figure 5: Results of algorithms on 2x2 intervals

μ	ρ	π	generate
$\{\underline{a}\}\{\}\{\overline{a},\underline{b}\}\{\bar{b}\}$	$\underline{x}^0\underline{y}^1\overline{x}^2\overline{y}^3$		$(\{\underline{a}\}\oplus\underline{x})^0$
$\{\overline{a},\underline{b}\}\{\bar{b}\}$	$\underline{y}^1\overline{x}^2\overline{y}^3$	$\{\}$	$(\{\overline{a},\underline{b}\}\oplus\underline{y})^1$
	$\overline{x}^2\overline{y}^3$	$\{\bar{b}\}\{\}$	$(\{\}\oplus\overline{x})^2$
	$\overline{y}^3$	$\{\}$	$(\{\bar{b}\}\oplus\overline{y})^3$

The first step will generate $(\{\underline{a}\}\oplus\underline{x})^0$ then the function is called recursively on the residual model $\{\overline{a},\underline{b}\}\{\bar{b}\}$ and residual commands $\underline{y}^1\overline{x}^2\overline{y}^3$ while memorizing $\{\}$. Thus, after having followed all the steps, the algorithm will have generated:

$$(\{\underline{a}\}\oplus\underline{x})^0(\{\overline{a},\underline{b}\}\oplus\underline{y})^1(\{\}\oplus\overline{x})^2(\{\bar{b}\}\oplus\overline{y})^3$$

By performing the distribution induced by $\oplus$ and by considering, to simplify the writing of the results, that for any element of the model m, for any command c we have $m\odot c=m$ (we take all the information in the model and the command only imposes its time), we get:

$$\underline{a}^0\overline{a}^1\underline{b}^1\overline{b}^3$$

This corresponds to the element of the studied box of the matrix exposed in the figure 5. Now, if we apply the same processing as performed on the model, and remove the time annotations, we get ,: $\{\underline{a}\}\{\}\{\overline{a},\underline{b}\}\{\bar{b}\}$, which corresponds to a *Meets* . By carrying out this transformation for all the elements of the matrix of the figure 5, we obtain a new matrix exposed in the figure 6.

It is easier to notice, in this new matrix, that the *Meets* and *Finishes* configurations are absorbing, whatever the algorithm and the commands, we can only produce configurations of these same types. On the other hand, *Before* type models are more interesting. They show that, depending on the algorithms, two different types of commands can generate different types of results: the interpreter, within the framework of the MidifilePerformer, is allowed to have an influence on the rendering. Moreover, we observe that, the more complex the algorithm becomes, the more potential influence on the result the type of the commands have. We will then talk about the *expressiveness* of the algorithm or the rendering function. Informally, this expressiveness is related to the number of possible configurations that a rendering function can produce for

		Before	Meets	Finishes
cas 0		Meets	Meets	Finishes
cas 1	Before	Before	Meets	Finishes
	Overlaps	Meets	Meets	Finishes
	During	Meets	Meets	Finishes
cas 2	Before	Before	Meets	Finishes
	Overlaps	Overlaps	Meets	Finishes
	During	Overlaps	Meets	Finishes
cas 2'	Before	Before	Meets	Finishes
	Overlaps	During	Meets	Finishes
	During	During	Meets	Finishes
cas 3	Before	Before	Meets	Finishes
	Overlaps	Overlaps	Meets	Finishes
	During	During	Meets	Finishes

Figure 6: Expressiveness

all models and commands, i.e. 2 for case 0, 3 for case 1, 4 for case 2 and 2', and 5 for case 3. More formally, if f is a rendering function of type $\mathcal{C}(T_1), \mathcal{C}(T_2) \rightarrow \mathcal{C}(T_3)$, its expressiveness will be defined as its co-domain:

$$Expressiveness(f) = \{r / \exists m, c, f(m, c) = r\}$$

We can focus on the expressiveness of a rendering function restricted to a particular model:

$$Expressiveness(f, m) = \{r / \exists c, f(m, c) = r\}$$

In other words, a performer may be concerned about the expressive potential that the *MidifilePerformer* may provide for the performance of a particular piece. Indeed, if the score mainly comprises *Meets* or *Finishes* type configurations, the algorithms will only allow a limited number of interpretations.

These considerations fully justify the option expressed at the end of the 3.3.1 section to transform some *Meets* configurations of the model to *Before*. This greatly amplifies the potential of interpretations.

4 Implementation of Rendering Functions

The algorithms given in the previous section presuppose the total knowledge of the data handled, namely the whole of the model and the commands, one also speaks of *static* or *post-mortem* analysis. As much as it is acceptable to know a score in advance, it is not conceivable to wait until the end of a performance to render it. The render function must therefore be *responsive*, it must provide results, *as soon as possible*, in response to commands. We then speak of *data flow* applications, which are a generalisation of producer/consumer or server/client problems. A MIDI keyboard can be seen as a server that produces MIDI events. At the end of the chain, we will find a synthesizer which will be assimilated to a client consuming MIDI events. In between are elements that are both consumers and producers. The data flow determines the graph from which we can identify the sources (keyboards) and sinks (synthesizers).

4.1 Data Flow: Push and Pull

Push and pull are two general techniques to implement data flow processing [7]. First, methods of type *push* transmit datas to clients as soon as they are produced. Second, *pull* type methods ask to producers datas when they are needed.

In *push* mode, the consumer registers to the producer in order to be alerted when a data is produced. In Java, this is particularly the case for obtaining MIDI events [5] or events coming from the graphical interface [6]. It is the consumer's responsibility to consume the resource as quickly as possible. If there are several consumers for the same producer, in general, the same resource, when produced, is distributed to all clients. For particular applications, Round-robin techniques can be used where the resources are given, one at a time, to each consumer.

In *pull* mode, the consumer explicitly requests a resource from the producer. This request can be blocking, it stops the computing unit of the client as long as the server has not produced a resource. In Java, this mode is used for the acquisition of sound samples coming from a microphone [7] or to obtain bytes from a stream (file, TCP connection etc.) [8] method.

It is possible to have access to a producer of type *push* and to be exposed as a new server of type *pull*. This technique requires memory space in order to store the resources awaiting a request from the client.

Conversely, it is possible to have access to a producer of type *pull* and to be exposed as a new server of type *push*. If the blocking wait for the resource is acceptable, this service transformation is costless.

In these two data flow type conversions, it is of course possible to perform a transformation of the received resource before transmitting it to the client. We thus find the functionality of a *map* applied to the flow of data.

A more delicate point appears when a node of the data flow graph has more than one antecedent: it has access to several producers. This is the case with the *combine* function with model and command chronologies. The choice of the types of the data streams is strongly influenced by the transformation that the node in question must perform. In our case, the events coming from the commands are the engine of the rendering function. As long as there are no new commands, there is no need to move forward in the model. It is this observation which made the implementation incline to use *pull* type streams.

4.2 The Chosen Implementation

Figure 7 shows the organization of the MidifilePerformer data flow acyclic graph (DAG).

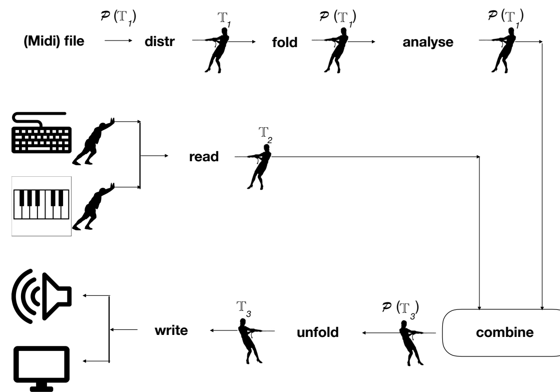

Figure 7: data flow of MidifilePerformer

The silhouettes on the graph's edges explain the type of data flow, *push* or *pull*. We have also noted

[5]interface `javax.sound.midi.Transmitter`
[6]methods `java .awt.Container.add***Listener`
[7]the method `javax.sound.sampled.DataLine.read`
[8]the `java.io.InputStream.read`

above these silhouettes the type of objects passing through the data stream. We observe that the whole is in *pull* mode except for the sources coming from the keyboards. Each of these inputs (*push*) writes in a shared memory of type *first in, first out* (FIFO). The *read* function reads this shared memory, in a blocking way, to provide a service of type *pull*.

The model chronology begins by reading the MIDI file, this is done in one block due to the organization of the file structured by tracks. The next function, *distr*, is an enumerator, it takes a few objects and distributes them, one by one, in *pull* mode. The function $fold$ compacts the events received when they have the same time, shaping the simultaneity concept. The *analyze* function organizes these concurrent event packets as described in the 3.3.1 section. These two functions, $fold$ and *analyze*, must memorize one event. For example, the function $fold$ must read one event too much to realize that its associated time does not correspond to the time of the packet it is building [9].

The central node of the graph corresponds to the function *combine* studied in the previous section. Its output provides event packets constructed by the combinator $\oplus$. These packets must be serialized, by the $unfold$ function, before being transmitted, one by one, to each of the sinks of the graph which know how to consume MIDI events.

All internal nodes of the graph are generic, they can be reused for types other than the MIDI objects. Note that it is the *write* function which will trigger all the read strings on the data streams of the *pull* type. These channels are blocked by the acquisition of an event on the keyboards.

4.3 Current State of the Software

The elements described in this article have been integrated into a graphical interface written in Java. The software code sources are available on *GitHub* [10]. The set is divided into three packages:

1. `core`. This package contains ten classes or interfaces, all generic with one exception. A first abstract class defines the flows. This class combined with the notion of event allows the development of the abstract class of chronologies. To set up the algorithms, generic types can rely on three interfaces defining equality on events, the notion of interval and the combination function ($\odot$). Two other interfaces make it possible to define the expected behavior of the generators at the origin of the production of events (which will have in particular instances for the MIDI inputs and for the alpha-numeric keyboard) and the expected behavior of the end consumers of the events (in fine a MIDI synthesizer or the display screen). Above all this, there are two concrete classes implementing a chronology allowing, on the one hand, to have the possibility of memoizing the last event consulted and, on the other hand, of memoizing all the events consulted. It is this last class which allows the recording of the events emitted by the interpreter in order to carry out a restitution *a posteriori*. Finally, a last class, the one which is not generic but which only has entry points (static methods) comprising genetic types, defines the various algorithms described in this article.

   This whole code is about 0.7 klocs [11].

2. `impl`. This second package gives concrete implementations of the previous package. All definitions are geared towards the MIDI specification. At this level, it is also defined a restricted subset of MIDI, having a simple syntax, in order to establish non-regression tests.

   This second package contains approximately 1.2 klocs. The base of test of non-regression is not included in this number, it comprises on its own 0.4 klocs.

3. `app`. The last package uses the concrete implementations in order to provide a user interface. There is approximately one class per each graphic component. The interface allows you to select the MIDI files to be interpreted, the inputs (MIDI keyboard or alpha-numeric keyboard), the outputs (synthesizer or screen), options (algorithm parameters, such as the filter). You can save the selected options in a

[9] We can make the analogy with lexical analyzers of the type *LL (1)*

[10] `https://github.com/scrime-u-bordeaux/MidiFilePerformer`

[11] kloc = thousand lines of code

configuration file which will be read at each launch. Finally, you can listen to the MIDI file at its own tempo, hear the interpretation you have made of it and save this interpretation as a MIDI file.

This last package has about 1.3 klocs.

The additional cost, in terms of lines of code, requested for the graphical interface is perhaps due to the graphical library used. However, these three packages remain in the same order of magnitude of volume.

5 Conclusions

We have described the internal structure of the MidifilePerformer application structured around model and command chronologies. The modification of the temporal associations between the elements of the model and of the commands introduces a notion of *expressiveness* which allows the interpreter to increase the possibilities of links between the notes while keeping the simplicity of the technique.

Compared to the previous version of the software, we have made an effort to unify the implementation of the model and command chronologies. These chronologies have been defined with generic tools that can be reused for data other than MIDI events.

The fact that the score is seen as a flow of data will allow us to develop more reactive interactions with the model. For example, we can consider that a second performer enters a performance by taking a specific channel from the MIDI file, this will call into question the pre-processing established in the 3.3.1 section, which will not be a problem if these treatments are done *on the fly* in the data stream.

Another perspective consists in generating an interactive scenario for the OSSIA score software[4][3] by modeling each note by a process. Thus, it would be possible to make modifications on the score, for example to manually simplify some musical excerpts in order to make them easier to perform, or on the contrary to give more control to the musician to increase the expressiveness, or to allow synchronization with other media engines to augment the score. Then the modified scenario would be executed by OSSIA score.

References

[1] James F. Allen. Towards a general theory of action and time. *Artificial Intelligence*, 23(2):123–154, 1984.

[2] The MIDI Manufacturers Association. The Complete MIDI 1.0 Detailed Specification. `http://www.freqsound.com/SIRA/MIDI\Specification.pdf`, 2014. [Online; accessed 05-May-2021].

[3] Jean-Michael Celerier. *Authoring interactive media : a logical & temporal approach.* Theses, Université de Bordeaux, March 2018.

[4] Jean-Michaël Celerier, Pascal Baltazar, Clément Bossut, Nicolas Vuaille, Jean-Michel Couturier, and Myriam Desainte-Catherine. Ossia: towards a unified interface for scoring time and interaction. In Marc Battier, Jean Bresson, Pierre Couprie, Cécile Davy-Rigaux, Dominique Fober, Yann Geslin, Hugues Genevois, François Picard, and Alice Tacaille, editors, *Proceedings of the First International Conference on Technologies for Music Notation and Representation - TENOR2015*, pages 81–90, Paris, France, 2015. Institut de Recherche en Musicologie.

[5] Arshia Cont. ANTESCOFO: Anticipatory Synchronization and Control of Interactive Parameters in Computer Music. In *International Computer Music Conference (ICMC)*, pages 33–40, Belfast, Ireland, August 2008.

[6] Irène A. Durand and Sylviane R. Schwer. A tool for reasoning about qualitative temporal information: the theory of s-languages with a lisp implementation. *Journal of Universal Computer Science*, 14(20):3282–3306, nov 2008.

[7] Conal Elliott. Push-pull functional reactive programming. In *Haskell Symposium*, 2009.

[8] Jean Haury. *Un répertoire pour un clavier de deux touches : théorie, notation et application musicale*, volume 11 of *Document numérique*. Lavoisier, 2008.