A robust algebraic framework for high-level music writing and programming

David Janin

A ROBUST ALGEBRAIC FRAMEWORK FOR HIGH-LEVEL MUSIC WRITING AND PROGRAMMING

David Janin
LaBRI CNRS & INRIA Bordeaux Sud-Ouest &
Bordeaux INP & Université de Bordeaux
351, cours de la libération
33 405 Talence cedex
janin@labri.fr

ABSTRACT

In this paper, we present a new algebraic model for music writing and programming. It is based on separating music object contents: what music they defined, and music object usage: how they can be combined. These are two orthogonal aspects of music representation/programming that should be kept separate although handled in a combined way.

From a mathematical point of view, music objects are modeled by means of some notion of tiled music graphs that can be combined by a single operator: the tiled sum. This operator is neither sequential nor parallel but both. The resulting algebraic structure is well studied in algebra: it is an inverse monoid.

From a programming point of view, our approach provides a high level domain specific language (DSL), the T-calculus, that is both reactive, hierarchical and modular. It is currently under implementation in the functional programming language *Haskell*.

From a representation point of view, various music examples are provided to show how notes, chords, melodies, musical meters and various kind of interpretation aspects can easily and robustly be modeled in this new formalism.

1. INTRODUCTION

1.1 From music programming languages

In the field of computer music, several music programming languages are available such as, to name but a few, the functional programming languages *Faust* [1] or *Euterpea* [2], the data flow programming languages *Max/MSP* [3] or *OpenMusic* [4], or a more imperative programming languages such as *CSound* [5] or *SuperColider* [6].

As a matter of fact, all these music programming languages *are* music representation formalisms. Indeed, every music program can be seen as sort of a music score that describe the music that can be played. Moreover, using music programing language necessarily induces a mental representation of the music pieces that can be defined and the way they can be defined. This implicit representation *does* influence the perception of the user [7]. Thus one may question the adequacy of a given programming language as a representation formalism.

More precisely, when seen as music representation formalisms, music programming languages must be abstract enough to allow the transcription of the composer's creative intentions. Music-oriented program constructs must be available and implementation details must be made, as much as possible, implicit. For instance, adequate user interface as in *OpenMusic* may hide programming language syntax, replacing it by higher level, graphic-based editors. However, representing the music that is encoded by a given program is not, in general, an easy task.

Somehow on the contrary, when seen as programming languages, these formalisms must be effective. The written music must be playable. For such a purpose, some implementation details need to be provided. Standard requirements of software engineering must also be satisfied. For instance, every complex program should result from the composition, duplication and transformation of simpler programs.

1.2 From music representations to programs

To some extent, every music representation formalism can also be seen as music programming languages. Classical western music notation is a perfect example of such a fact. Numerous music editors allows to define music scores that, in turn, can be encoded into computer objects that can be played.

However, even though musical objects are rather well defined in music sheets, western notations do not provide a rich set of composition and transformation methods. Dedicated to written music, these notations suffice to write music, but not necessarily to create it. Even worse, the more or less adhoc functionalities offered by music editors may be musically ill-defined.

For instance, the standard cut, paste and insert transformations, essentially inherited from text editors, are inadequate because they often break the global consistency of the music. Indeed, inserting a sequence of notes in a score necessarily pushes the rest of the score regardless of the underlying musical meter.

Somehow paradoxically, music composition operators are well defined in music programming language. However, in a programing language, relevant music concepts may be lost in non musical technicalities. Specialized technician may be required to bridge the gap between composer ideas and their programming realisation.

1.3 Model based approach

One way to handle both music representation necessities and software engineering requirements is by deriving both of them from a unified model of musical objects with well understood algebraic properties.

Indeed, every music programming language, every music design software, as well as every music representation formalism induces a more or less implicit *music algebra* that defines both the basic objects that can be used and the combinators that allow to build complex music objects from simple ones. In the absence of a unified model, it is very likely that these algebras will be incoherent. In a model based this cannot happen as illustrated in Figure 1.

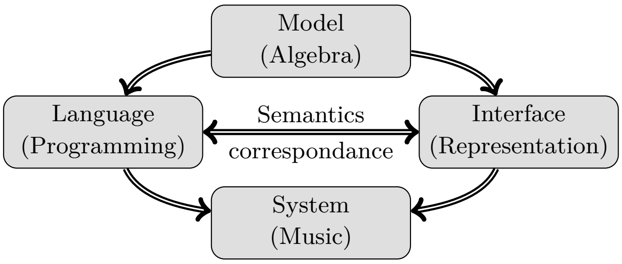


Figure 1: Model based approach.

Indeed, every functionality, be it defined at the programing level or at the representation level, necessarily derives from the algebraic model. Any (mental or explicit) representation induced by the usage of these functions converges to a single and coherent representation of the elements of this algebra. A correspondance between music programs and music representations becomes possible.

In the long term, this may even leads to the definition of a well founded graphic-based programming interafce, henceforth offering an easier access to programming techniques.

1.4 Known approaches

Technically, following such an approach leads to the design of a Domain Specific Languages (DSL). These languages are high-level programming languages dedicated to a specific application domain. They provide no more and no less than the necessary high-level constructs relevant to the underlying application domain [8]. In music programming, languages such as *Faust* [1] or *Euterpea* [2] are examples of DSLs.

In most of the proposed languages, musical objects are mainly combined by means of two operators: a sequential composition that allows to play two musical objects one after the other, and a parallel product that allows to start in parallel two musical objects. Hudak's notion of polymorphic temporal media [9] makes this algebra explicit. It allows to reason about programs and provides a better understanding of program's semantics.

However, it has already been argued [10] that the algebra induced by sequential and parallel operators, though rather natural at the DSP level provided by a language such as *Faust*, induces implementation oriented point that may not be that convenient in the case of high-level musical design.

Using the parallel (or rather fork) operator induces a "forward vision" of music writing, from the past to the future. It merely amounts to decide at any time *what comes next*. Musical metrics, synchronization points, visualized by bars in classical western music notations, quite disappear under such a unidirectional view of music writing [11]. As a matter of fact, typical music constructs necessitates a "backward vision", from the future to the past, that allows to decide at a given time *what comes before*.

Cadences such as $II^m/V^7/I$ or $II^m/I^7_\sharp/I$ are typical exemples of such phenomena. Resolving on a first degree constitutes a goal in the future. A cadence is a way to reach such a goal. It is thus implicitly understood as a construction from the future to the past. Another exemple is the anacrusis. Aiming at introducing a given note on a strong beat, an anacrusis is positioned in a backward way, depending on its length. It can be argued that such a phenomenon cannot be modeled properly with a forward point of view [10, 12].

1.5 Contribution and structure of the paper

Our goal is thus to define such a music algebra from which we derive both a programming language and a representation formalism.

For this purpose, aiming at relaxing the "forward" point of view induced by the sequential/parallel music algebra we have developed the notion of tiled modeling [13, 14] and tiled programming [15]. Both sequential and parallel composition operators are eventually merged into a single one : the tiled composition operator [12, 11]. This offers a higher-level point of view over the usual sequential/parallel music algebra.

In this paper, we present the latest development of this new music algebra and we eventually provide various and explicit music modeling examples. Doing so, aside the mathematical robustness of our approach that is already detailed in former presentations, we aim at illustrating the relevance of this approach for music modeling.

In Section 2 we describe a modeling example that essentially details what representation models we aim at achieving. How these models can generated from basic elements is detailed in the remaining sections.

In Section 3 we describe the formal model of tiled music graphs that are equipped with a single operator: the tiled composition. Properties of the resulting algebra that are relevant for music modeling are presented.

Then, in section 4, we show how these music graphs can simply be generated by means of music graph expressions. Music modeling examples are provided throughout all sections.

2. A MODELING EXAMPLE

Before getting into a detailed description of tiled models and programs, we provide in this section a first modeling example that illustrates most of the musical features we aim at capturing. The way such a model may actually be built, step by step, is described throughout the next sections.

2.1 Debussy's first arabesque

Debussy's first arabesque is a typical example of a music structure which is difficult to model by means of tree-based (or syntactic) modeling approach [16]. Indeed, as illustrated in the 91th and 92th bars of Debussy's first arabesque (see Figure 2), four voices, with distinct rhythmic structures, are interleaved. They induce rhythmic trees that have incompatible shapes and thus cannot be shared properly [17].

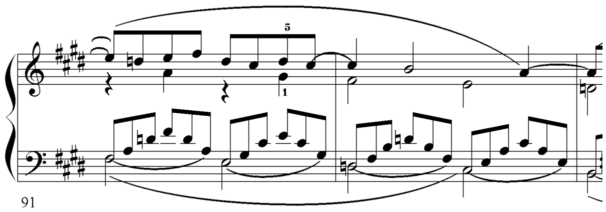


Figure 2: Two bars of Debussy's first arabesque.

Our modeling approach amounts to describe this piece of music by means of a music graph: a directed labeled graph vertices modeling instants in time and with edges modeling musical elements such as rests, notes or chords.

More precisely, the first half of Debussy's 91th bar is depicted in Figure 3. The various features that are appearing in this figure can be be detailled as follows :

- *vertices*: depicted by bullets (•); vertices are modeling instant in time that are called synchronization points,
- *directed edges*: depicted by arrows (• ⟶ •); edges are modeling basic musical such as rests, notes or chords that last for some duration : the time that elapse between the two synchronization points they relate,
- *upper or left edge labels*: depicted by (blue) positive rational numbers that appear either on the left or above the middle of each edge; these edge labels are modeling durations,
- *lower or right edge labels*: depicted by (red) strings that appear either on the right or below the middle of each edge; these edge symbols are giving the nature of rests, notes or chords.

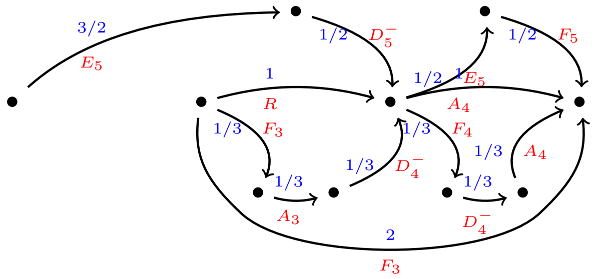


Figure 3: A graph model of part of the 91th bar.

Observe that the first E_5 is of duration 3/2 since the initial eight note E_5 is actually linked with a quarter note in the 90th bar, much in the same way the first A_4 quarter note in the 93th bar is linked with the quarter note A_4 at the end of the 92th bar.

On this graph, R indicates no notes, that is, a rest. One can also observe that all D have been written D^-. This models the fact that, as indicated in the key signature, all these notes must be lowered by one semitone, from $D^\sharp$ down to natural D.

A remarkable property of this graph model is that, if one counts a positive duration when traversing an edge forward and if one counts a negative duration when traversing an edge backward, then, the resulting sum of durations along any cycle always equals zero. Indeed, this follows from the fact that vertices are modeling instants in time henceforth every cyclic traversal amounts to go back and forth in time until the initial point is eventually reached.

Remark. One may observe that some musical elements are missing in this modeling. Indeed, neither the musical meter, nor the key, nor even the links indicating musical phrases are described.

However, as opposed no note duration or names, these attributes apply to groups of musical objects. We thus need a way to specify these groups or, as another way to say it, we need the algebraic tools that allow to generate these groups.

It follows that the modeling of group attributes is postponed to the last section when the music algebra will be defined.

2.2 Alternative graph representations

Of course the graph depicted in Figure 3 is not easily readable by humans. Another visualisation, more oriented towards human, is depicted in Figure 4.

In this figure, every vertex has been scattered along a vertical dashed line. These lines have also been labeled by their distance, expressed in quarter note durations, from the beginning of the bar. Clearly, such an illustrative vertex labeling can easily be computed from the duration labeling the edges, as soon as one vertex is chosen as the origin.

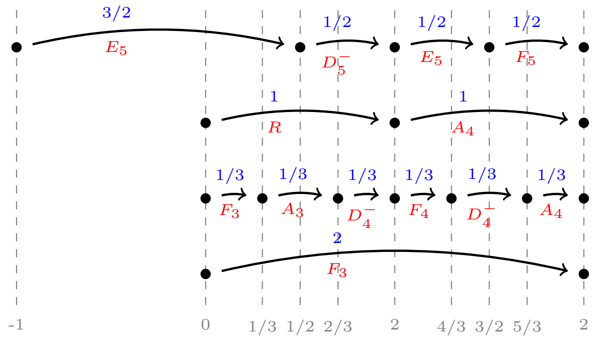


Figure 4: Scattered view of the same model.

Remark. Of course, other representations are possible such as, for instance, presentations based on cyclic or spiral shaped timelines. The reader should not make the confusion between the underlying mathematical models that are graphs, possibly unreadable, and their possible representations that may take various forms more readable.

3. TILED MUSIC GRAPHS

The models we aimed at building, music graphs, have been depicted in the previous section. As such, they almost form a music representation formalism [1] . The question we adress from now and throughout the remaining sections is how to generate such representations.

The resulting algebra is presented in two step. The first step is only concerned with generating timed graphs. It is presented in this section.

Then, it can be shown that such a modeling can simply be extended to music graphs by associating values to timed graph edges. As a result, we obtain an algebraic language for defining musical objects. Such a point of view is presented in the next section.

3.1 Timed graphs

We first aim are generating timed graphs, that is, a *directed acyclic graphs with labeled edges* with vertices representing synchronization points and edges labeled by the duration representing yet unspecified musical objects or rests. Examples of basic timed graph are depicted in Figure 5. They can be detailled as follows. In (5a), two musical objects of respective duration a and b are launched in parallel starting at the same time. In (5b), similar musical objects are played independently. There is no temporal dependency between them. In (5c), two musical objects are finishing at the same time. In all these figures, there is no knowledge of the respective values of a and b that are presumably distinct.

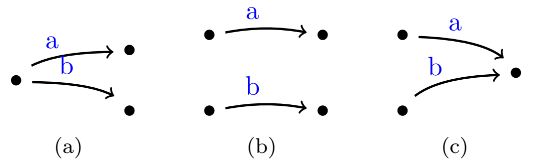


Figure 5: Basic timed graphs.

3.2 Local unambiguity

By assumption, vertices are synchronization points in time. It follows that two musical objects with the same durations that are launched at the same time eventually are reaching the same synchronization point. Borrowing the vocabulary from automata theory, timed graphs are both *deterministic*.

Symmetrically, two musical objects with the same duration that end at the same time necessarily start from the same syncrhonization point. Timed graphs are also *co-deterministic*. In other words, timed graphs are bi-deterministic graphs or, as another way to say it [18], *locally unambiguous*.

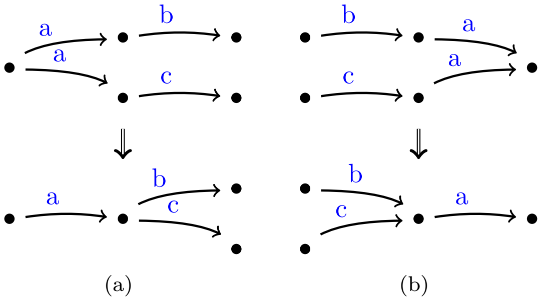


Figure 6: From ambiguous to unambiguous timed graphs.

It is shown in [18], there is a simple way to transform arbitrary directed graphs into its greatest locally unambiguous image. Indeed, it just amount to merge every pair of edges with the same duration label that share a common origin (see (6a)) or target (see (6b)).

Of course, this merging process needs to be repeated until the resulting graph is locally non ambiguous. Doing so, it may happend that acyclicity is lost; a directed cycle may appear. In this case, the timed graph is considered to be erroneous. In a derived programming tool, a design error is raised.

3.3 Synchronization attributes

Timed graphs are extended by two synchronization points: two distinguished vertices respectively called the input and the output root of the timed graph.

[1] up to the modeling of group attributes that we have postponed.

Resulting graphs are simply called birooted graphs or, to fit our application perspectives, birooted timed graphs.

Examples of birooted timed graphs are depicted in Figure 7 where input roots are depicted by (ϒ) and output roots are depicted by (↓).

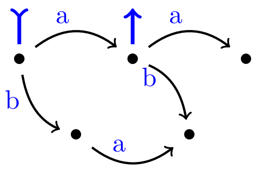


Figure 7: Two birooted timed graphs.

3.4 Tiled composition

These graphs can then be combined by means of the tiled composition. More precisely, these specified input and output roots allows to combine two musical objects [2] by gluing the input root of the first one with the input root of the second: this is the *synchronization step*. The local ambiguity that may results from these gluing is then removed following the bi-determinization process depicted above: this is the *fusion step*. The input (resp. output) root of the first object (resp. the second) is kept as the input root (resp. the output root) of the second object.

The birooted graph resulting from the composition of the two graphs depicted in Figure 7 is depicted in Figure 8 below.

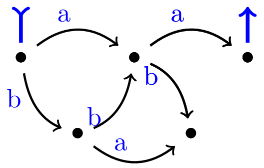


Figure 8: The result of a tiled composition.

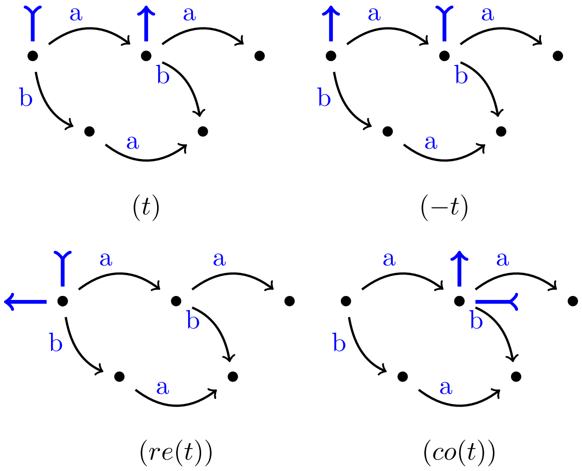
It is know from inverse semigroup theory [19] that the resulting composition is associative. Since the single vertex graph with equal input and output root is clearly a neutral element for this composition, the resulting graph is known in algebra as a monoid. Moreover, it can also be shown that this monoid is an inverse monoid [20] (see also[18]).

From now on, such a composition is denoted additively, that is, the tiled composition of two tiled timed graphs t_1 and t_2 is denoted by $t_1 + t_2$. The single vertex graph with equal input is denoted by 0 and we clearly have $t + 0 = t = 0 + t$ for every tiled timed graph t.

3.5 Inverse, reset and coreset

The "inverse" arising from the underlying monoid is also denoted additively. In other words, for every tiled timed structure t, it is denoted by $-t$. It is just obtained from t by permuting the input and output roots, without changing the direction of the music.

This allows to define the difference $t_1 - t_2$ between two music graphs as the sum $t_1 + (-t_2)$. Then, we define the *reset* of t by $re(t) = t - t$ and the *coreset* of t by $co(t) = -t + t$. These remarkable elements are depicted in Figure 9 below.



(t) $(-t)$

$(re(t))$ $(co(t))$

Figure 9: Resets and coresets.

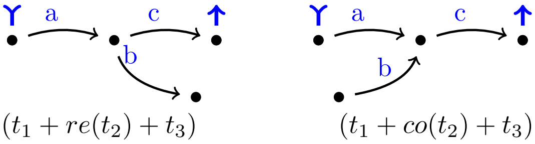
Observe that, unless the music graph t is the graph of the zero duration rest 0, none of these expressions equals zero. However, as detailed in inverse semigroup theory, the reset and coreset defines *local zeros* as made explicit in the following equation that are always satisfied:

$$t = re(t) + t \text{ and } t = t + co(t)$$

3.6 Induced parallel composition

When building complex time structures (or later musical structures), a typical usage exemple of reset and coreset primitives are parallel insertions.

Indeed, given three tiles t_1, t_2 and, resp., t_3, simply denoting single edges of duration a, b and, resp., c as depicted in Figure 10.



$(t_1 + re(t_2) + t_3)$ $(t_1 + co(t_2) + t_3)$

Figure 10: Parallel insertions.

A construction of the form $t_1 + re(t_2) + t_3$ inserts a copy of t_2 between t_1 and t_3 without altering the synchronization of t_1 and t_3. More precisely, it amounts

[2] yet just birooted timed graphs, but the composition of general musical objects defined later in the text just follows the same rules,

to start t_2 at the end of t_1 in parallel with t_3, with all possible overlaps and merges allowed by tiles.

Symmetrically, a construction of the form $t_1+co(t_2)+t_3$ still inserts a copy of t_2 between t_1 and t_3. However, in this case, both t_1 and t_2 ends at the same time, before t_3 starts.

In both cases, synchronization between t_1 and t_3 are just the same as they would be in the sum $t_1 + t_3$ so both cases describe sort a parallel insertion.

These constructions offered by the reset and the coreset primitives constrast with the standard, string-based, insertion primitives that are available in most softwares for graphical music editions; classical insertions that would "push" appart t_1 and t_3.

3.7 Additional time equations

A carefull reading of the example depicted in Figure 8 above shows that more vertices gluing have been performed than those that were strictly necessary. This comes from the fact that we have additionally applied the equation

$$a = bb$$

that says that the duration of two bs just equals the duration of one a.

In other words, under the equation $a = bb$ two succesive b-labeled edges starting from the same starting synchronization point than an a-labeled edge eventually reach the same ending synchronization point. With a view towards application in music, such a feature allows to define standard timed graphs with length measured in hole notes, half notes, quarter notes, etc. . . , these relative lengths being easily defined by such kind of equations.

Observe that the graph depicted in Figure 3 just follows these rules. The semantics of duration labels such as 1, 1/2, 1/3, etc., . . . has implicitly been defined by such a kind of equations. For instance, two successive edges of duration 1/2 eventually reach the same synchronization point than a single edge of duration 1.

4. THE RESULTING MUSIC ALGEBRA

Tiled timed graphs are turned into musical graphs by adding additional attributes to edges. In the proposed frameworks, these edge attributes can be sets of notes, possibly with some more attributes denoting instruments, tracks, velocity, etc.. . . Additionally, global (or group) attributes can also be defined over tiled musical graphs in order to describe some expressive features such as legato.

In this presentation, we restrict ourselves to the simpler case edge attributes are sets that are simply combined by union. Observe that such a set based modeling of edge attributes is already implicitly used in Figure 3 where rests are modeled by the attribute 0 that denotes the empty set of notes.

It can be shown that adding edge labels from a lattice does preserve the inverse monoid structure. In view of our modeling perpectives, we show in this section how this theory can be put in practise.

The resulting labeled tiled timed are from now on simply called music graphs.

4.1 Elementary music graphs

Elementary music graphs are either rests, denoted by R or a single note, of the form X^e_i where X is a pitch class (e.g. A, B, C, etc. . .), i is an octave, and e is a possible alteration. For instance $C^\sharp_4$ denotes C sharp on the fourth octave.

By default, all notes or rests duration equals one quarter. In other words, $C^\sharp_4$ as above actually denotes a quarter note. However, every note or rest (and later every score) can be stretch by some rational factor. Such a stretch is denoted by a left multiplication. For instance, the expressions $2 * D^\flat_5$ denotes the half note D flat on the fifth octave. Similarly, the expressions $1/2 * E_3$ denotes the eight note E on the third octave.

Of course, choosing quarter notes as unit duration is arbitrary. Our choice follows from the fact that, quite often, quarter notes represents one beats. Following english duration naming rules, it would certainly makes sense to use the duration of whole notes as duration unit. But this would just amounts to multiply all written melodies by 1/4 so this can be done in the last moment. In other words, the naming of duration can be left to the user.

By convention, single fractions are also seen as rests. For instance, the notation 2 is equivalent to $2 * R$. As a particular case, the notation 0 stands for the rest of duration zero. This implies that $d * 0 = 0$ for any stretch factor d.

4.2 Synchronized sums of music graphs

The tiled composition arising from the underlying monoid is written additively not to be confused with the stretch. In other words, given two music graphs t_1 and t_2, we denote by $t_1 + t_2$ the synchronization of the first music graph t_1 with the second one t_2.

Somehow as expected, for every music graph t, we have $t + 0 = 0 + t = t$.

For instance the following expressions denotes the beginning of a little waltz.

$$1/2 * (2 * C_4 + D_4 + 2 * E_4 + G_4 + 2 * E_4 + D_4 + 3 * E_4)$$

when played in 3/8. The resulting graph is depicted in Figure 11 below.

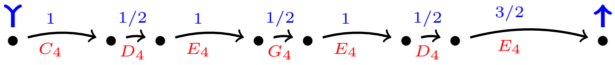


Figure 11: A little waltz.

Observe that the stretch operation distribute over the sum. For instance, the expression $1/2*(2*C_4+D_4)$ may as well be denoted by the equivalent expression $(C_4 + 1/2 * D_4)$.

4.3 Generalized product

The stretch of a music graph t by a factor d is thus expressed as $d * t$. Viewing d as a rest of length d, this means we allow certain product of musical graphs. This construction may be generalized. In fact, for every music graph t_1 and t_2, we put

$$t_1 * t_2 = re(|t_2| * t_1) + |t_1| * t_2$$

where, for every music graph t, we write $|t|$ for the elapsed time from the input root of t to the output root of t.

Using such a generalized product, we can thus add a bass line to our little waltz, by putting

$$C_3 * (2 * C_4 + D_4) + G_2 * (2 * E_4 + G_4) + \dots$$

The resulting music graph is depicted Figure 12.

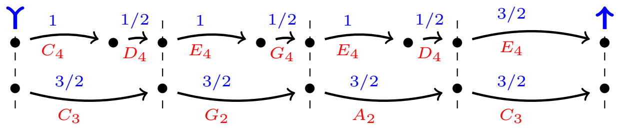


Figure 12: Adding a bass line.

4.4 Debussy example: the return

As an illustration of the tiled music algebra, we can build step by steps the musical graphs depicted in Figure 3. As already detailed above, the basic elements are notes with associated durations. Then these elementary tiles can be combined via sums, stretches, possibly taking inverses, resets or coresets.

For instance, the soprano voice of the 91th bar of the arabesque can then be defined by

$$v_1 = 1/2 * (2 + co\left(3 * E_5 + D_5^-\right)) + E_5 + F_5 + D_5^- + C_5 + re\left(D_5^- + 3 * C_5\right) + 2)$$

In this construction, a coreset (resp. a reset) is used to model the initial E_5 (resp. the final C_5) that starts before the beginning of the bar (resp after the end of the bar).

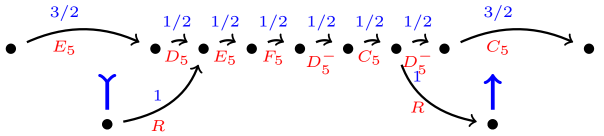


Figure 13: The soprano voice v_1 of the 91th bar.

The resulting music graph is depicted Figure 13.

The second voice, an alternation of rests and quarter notes, can simply be defined by $v_2 = 1 + A_4 + 1 + G_4$. Then, the two combined voices can be defined by $re(v_1) + v_2$ or, equivalently, $v_1 + co(v_2)$, among many other equivalent expressions since the distance between their input and output roots are both equal to 4.

The result of the combined first and second voices is depicted in Figure 14. Again, distances from the input root are labeling the vertical lines.

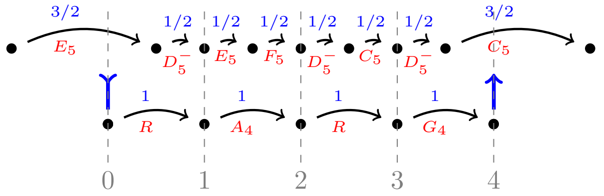


Figure 14: The combined first and second voices of the 91th bar in scattered representation.

Clearly, the entire 91th bar can be modeled this way, as well as the combination of all bars, till the complete Arabesque is modeled.

Remark. Compared to the tree-shaped model approach proposed in [16], one may observed that the terms defining voices are somehow similar : as terms there are indeed tree shaped. However, our graph based approach allows easily to combine time structures that have different shapes. Moreover, the parallel operators just follows from the inverse semigroup structures that provide inverses, resets and coresets.

Somehow implicitly, the tree-shaped terms defining birooted musical graphs are factorized by the relevant notion of musical equivalence (aka two terms are equivalent when they defined the same birooted graphs). The fact this equivalence is a congruence, that is, it remains stable under sum, is by no mean a trivial property.

4.5 Group attributes

We are now ready to address the question of group attributes. The main idea is that a group attribute such as links or time signature can be seen as a special edge attribute that is distributed over a group of edge. Since edge attributes are sets of values, it makes no difficulty to extend the possible values on edge by any additionel set of marks. The resulting algebra is essentially the same.

At the programmatic level it is convenient to specify only once any group attribute. This is simply done by adding another functions that add the attribute values to all edges of its tile argument. Assume that such a function is $att(t, v)$. Then, given two musical tile t_1 and t_2, we may specify that t_1 will be played with a 3/4 meter and, afterward, t_2 will be played with 4/4 meter just by the sum $att(t_1, 3/4) + att(t_2, 4/4)$.

At the representation level, a construction of the form $att(t, v)$ can be visualized by adding some background color on the underlying group of edges making explicit the "region" that have been affected by the group attribute value v.

Of course, in a sum of the form $att(t_1, v) + att(t_2, v)$ one may argue that the two underlying groups are implicitly merged since they seem to share the same group value. However, at the implementation level, the syntactic distinction between the first and the second group can simply be achieved by extending the value v by a single *id*, automatically generated, that refers uniquely to a given call to the function *att*. Such an encoding, invisible to the user, prevents any possible confusion.

Remark. It can be the case that some group attribute values cannot overlap. For instance, an edge labeled by both a 3/4 and a 4/4 meters would make no sense.

This amounts to say that we must forbid edge attribute that contains two incompatible group attribute. It occurs that such a property is stable under composition hence it forms a monoid ideal. At the algebraic level, all these tile can be merged [3] into a forbidden tile $\perp$. Forbidding music tile with cyclic underlying graphs is just handled the same way since the tile property "having a cycle" is also preserved by composition.

5. CONCLUSIONS

Starting from a fairly simple notion of music model, defined by means of instants in (symbolic) time that are related with elementary music objects, we have shown that adding synchronization points leads to a fairly robust algebra: an inverse monoid. This algebraic modeling provides, thanks to its richness, most of the music construct a composer may need, either as primitive constructs such as the tiled composition, or as a derived constructs such as the reset and the coreset functions.

All algebraic expressions and their corresponding graphical views that appear throughout our presentation proves that, based on the underling algebra, a robust correspondance between music programs and music representations can be developed. To some extent, the tasks of playing music or representing music have many common features that are made explicit in this approach.

The DSL induced by such an approach is currently under developpement. Embedded into a high level functional programming language such as *Haskell* and the various libraries available such as Euterpea [2] and UISF [21], it allows to inherit from its elegance and expressive power [12, 11, 22]. Design mistakes can be controlled both at the static level via the underlying type system and, at the dynamic level, thanks to the lazy evaluation mechanism of *Haskell*, via the evaluation of time or shape constraints.

Of course, this project is still at an experimental stage. Designing a new model and developing the related methods and tools necessarily take time [23], especially when we aim at revealing and exploiting the robustness of the underlying mathematical framework that may appear.

[3] technically, this amounts the take the quotient of music tiles by this ideal; such a quotient, well studied in semigroup theory, is called a Rees' quotient.

6. REFERENCES

[1] Y. Orlarey, D. Fober, and S. Letz, "Faust: an efficient functional approach to DSP programming," in *New Computationals Paradigms for Computer Music.* Editions Delatour France, 2009.

[2] P. Hudak, *The Haskell School of Music : From signals to Symphonies*, Yale University, Department of Computer Science, 2013.

[3] A. Cipriani and M. Giri, *Electronic Music and Sound Design - Theory and Practice with Max/Msp.* Contemponet, 2010.

[4] J. Bresson, C. Agon, and G. Assayag, "Visual Lisp / CLOS programming in OpenMusic," *Higher-Order and Symbolic Computation*, vol. 22, no. 1, 2009.

[5] B. Vercoe, "Csound: A manual for the audio processing system and supporting programs," MIT Media Lab, Tech. Rep., 1986.

[6] J. McCartney, "SuperCollider: a new real time synthesis language," in *Int. Computer Music Conference (ICMC)*, I. C. M. Association, Ed., 1996.

[7] M. Malt, *La représentation dans le cadre de la composition et de la musicologie assisteées par ordinateur.* Université de Strasbourg, 2105, habilitation à Diriger les Recherches.

[8] P. Hudak, "Keynote address - the promise of domain-specific languages," in *Proceedings of the Conference on Domain-Specific Languages (DSL)*, 1997.

[9] ——, "A sound and complete axiomatization of polymorphic temporal media," Department of Computer Science, Yale University, Tech. Rep. RR-1259, 2008.

[10] D. Janin, "Vers une modélisation combinatoire des structures rythmiques simples de la musique," *Revue Francophone d'Informatique Musicale (RFIM)*, vol. 2, 2012.

[11] P. Hudak and D. Janin, "From out-of-time design to in-time production of temporal media," LaBRI, Université de Bordeaux, Research report, 2015.

[12] ——, "Tiled polymorphic temporal media," in *Work. on Functional Art, Music, Modeling and Design (FARM).* ACM Press, 2014, pp. 49–60.

[13] F. Berthaut, D. Janin, and B. Martin, "Advanced synchronization of audio or symbolic musical patterns: an algebraic approach," *International Journal of Semantic Computing*, vol. 6, no. 4, pp. 409–427, 2012.

[14] D. Janin, F. Berthaut, and M. Desainte-Catherine, "Multi-scale design of interactive music systems : the libTuiles experiment," in *Sound and Music Comp. (SMC)*, 2013.

[15] D. Janin, F. Berthaut, M. DeSainte-Catherine, Y. Orlarey, and S. Salvati, “The T-calculus : towards a structured programming of (musical) time and space,” in *Work. on Functional Art, Music, Modeling and Design (FARM).* ACM Press, 2013, pp. 23–34.

[16] F. Lerdahl and R. Jackendoff, *A generative theory of tonal music*, ser. series on cognitive theory and mental representation. MIT Press, 1983.

[17] D. Rizo, “Symbolic music comparison with tree data structures,” Ph.D. dissertation, Universidad de Alicante, November 2010.

[18] D. Janin, “Inverse monoids of higher-dimensional strings,” in *Int. Col. on Theor. Aspects of Comp. (ICTAC)*, ser. LNCS, vol. 9399, 2015.

[19] M. V. Lawson, *Inverse Semigroups : The theory of partial symmetries.* World Scientific, 1998.

[20] J. Stephen, “Presentations of inverse monoids,” *Journal of Pure and Applied Algebra*, vol. 63, pp. 81–112, 1990.

[21] P. Hudak, D. Quick, M. Santolucito, and D. Winograd-Cort, “Real-time interactive music in haskell,” in *Work. on Functional Art, Music, Modeling and Design (FARM).* ACM, 2015, pp. 15–16.

[22] S. Archipoff, “An efficient implementation of tiled polymorphic temporal media,” in *Work. on Functional Art, Music, Modeling and Design (FARM).* ACM, 2015, pp. 25–34.

[23] W. Thomas, “Logic for computer science: The engineering challenge,” in *Informatics - 10 Years Back, 10 Years Ahead.*, ser. LNCS, vol. 2000. Springer, 2001, pp. 257–267.