Modéliser la structure musicale à l'aide des tuiles hiérarchiques

Alice Rixte, David Janin

MODÉLISER LA STRUCTURE MUSICALE AVEC LES TUILES HIÉRARCHIQUES

Alice Rixte
LaBRI
Université de Bordeaux
alice.rixte@u-bordeaux.fr

David Janin
LaBRI, Bordeaux INP
Université de Bordeaux
janin@labri.fr

RÉSUMÉ

Dans cet article nous utilisons la notion de graphe de scène, bien connue en informatique graphique, pour répondre au problème de la représentation musicale en informatique. Le modèle résultant, associé à un véritable langage de modélisation, doit permettre de décrire avec souplesse et flexibilité non seulement des partitions de musique mais aussi de rendre compte de la structure des oeuvres décrites. Ce modèle, son langage et sa sémantique sont illustrés ici à travers une modélisation détaillée de la comptine musicale *Do, ré, mi, la perdrix*.

1. INTRODUCTION

Contexte et motivations

Le problème de la représentation musicale en informatique est indissociable de l'informatique musicale elle-même. Les chercheurs tels que Lejaren Hiller [**?**] ou André Riotte [**?**] ont très tôt fait le constat que la partition musicale telle que développée en occident depuis plusieurs siècles ne constitue qu'une bien mauvaise réponse à cette problématique.

En effet, la partition telle qu'elle apparaît dans les livres de formation musicale en occident vise surtout à permettre une lecture rapide de la musique à jouer ainsi que la synchronisation des différents interprètes. En particulier, la partition musicale échoue largement à rendre compte de la structure de la musique qu'elle décrit. Ce n'est que la culture des interprètes qui leur permet d'expliciter, pour l'auditeur, une partie de cette structure implicite, pourtant essentielle à l'intelligibilité de l'œuvre.

Au contraire, l'ordinateur, qui lui n'a aucun problème de rapidité de lecture, doit disposer d'une représentation structurée de la musique, aussi bien pour l'aide à l'analyse [**?**] que pour l'aide à la composition [**?**], ainsi que pour son utilisation sur scène dans des dispositifs toujours plus complexes d'aide à la performance musicale.

En analyse musicale classique, cette structure apparaît à travers les nombreuses annotations ajoutées à la partition pour rendre compte des unités mélodiques, des structures harmoniques, et des formes multi-échelles de la musique qui y est décrite. En composition instrumentale, la partition n'est que le résultat final, destiné aux musiciens, d'un long processus de conception, qui s'est attaché au moins autant sinon plus à définir les formes de l'œuvre que son détail note à note.

En performance musicale assistée par ordinateur, la notion de *partition dynamique interactive*, c'est-à-dire la représentation de scénarios d'interactions complexes entre ordinateurs et interprètes [**?**], dépasse le cadre de la musique puisqu'elle s'insère dans la vaste problématique des interactions humain-machine en temps-réel.

Bien entendu, cette dernière remarque appelle à la modestie. La recherche d'une représentation informatique adequate de la musique ne sera bien évidemment pas résolue ici. Au mieux, le modèle présenté dans cet article nous permet de faire un pas de plus dans cette direction.

Contribution

Cet article développe un modèle algébrique de représentation musicale, destiné à être codé dans les ordinateurs, qui permet de rendre compte à la fois de la partition d'une œuvre musicale, mais aussi de sa structure.

Pour le concevoir, nous nous appuyons sur une définition résolument contemporaine de la musique comme étant *l'articulation du son dans le temps et dans l'espace* [1].

Quels sons ? Quelles échelles de temps ? Quelles dimensions d'espace ? Notre proposition est de ce point de vue agnostique, la notion informatique de polymorphisme de types nous permettant de nous abstraire de toute définition préalable d'échelles de temps ou de paramètres musicaux.

Une première interprétation de la définition ci-dessus nous conduit, avant tout, à permettre la description d'objets sonores placés dans un certain temps et un certain espace.

Comme ces objets peuvent être positionnés les uns par rapports aux autres via des transformations simples (toujours reversibles) de l'espace-temps sous-jacent, notre modèle autorise ainsi un codage relatif, algorithmiquement efficace, de la position de ces objets.

En s'autorisant à regrouper ces objets pour leur appliquer localement des transformations plus complexes (parfois irréversibles), nous pouvons aussi rendre compte de l'aspect hiérarchique de la musique représentée. Un objet

1 . Une définition proposée par le compositeur J.-L. Agobet

[noindent, staffsize = 16]doremi-score.ly

Figure 1. Partition de la comptine "Do ré mi, la perdrix"

musical peut donc aussi être défini comme la transformation d'un objet musical similaire ou comme une combinaison d'objets musicaux plus élémentaires.

Notre proposition est résolument orientée vers la mise au point d'un modèle de représentation interne à l'ordinateur. La question de la définition d'une interface humain-machine (IHM) pour la définition et l'édition de ce modèle est donc temporairement mise de côté même si tout porte à croire que cette question pourra être résolue.

En effet, notre proposition s'appuie largement sur la notion de *graphe de scène* [**?**], omniprésente dans le domaine de l'informatique graphique. Cette connection forte avec l'informatique graphique devrait permettre de définir une telle IHM.

En particulier, la notion de construction d'objet complexe à partir d'objets simples peut être reliée à cette fonctionnalité qu'offrent la quasi-totalité des logiciels d'édition graphique de définir un nouvel objet graphique complexe en « collant ensemble » des objets graphiques plus simples.

Une implémentation de ce modèle en *Haskell*, un langage fonctionnel typé avec types polymorphes, nous permet aussi de définir un langage déclaratif dédié (Domain Specific Language) qui apparaît alors comme une véritable interface de programmation (Application Programming Interface) permettant de générer et manipuler nos modèles.

Autrement dit, le modèle proposé ici, tout en trouvant sa source dans une définition particulièrement abstraite de la musique, et en s'appuyant sur des concepts mathématiques tout aussi abstraits, est déjà largement mis en œuvre en informatique à travers une implémentation robuste.

Bien entendu, la nécessité d'illustrer ce modèle nous conduit largement à l'opposé du polymorphisme annoncé. Le modèle présenté dans cet article est en effet décliné à travers un cas d'école : la modélisation de la comptine *Do ré mi, la perdrix*, dont la partition est donnée par la figure **??**. Sa simplicité et sa forte structuration nous permet d'illustrer la majorité des concepts évoqués ci-dessus. Dans ce cas, le temps musical est exprimé en nombre de battements, et l'espace musical est réduit aux notes de la gamme de Do majeur.

L'applicabilité de ce modèle à d'autres contextes musicaux est donc, dans cet article, laissé à l'appréciation de la lectrice ou du lecteur.

Travaux connexes

En informatique musicale

Les travaux présentés ici se situent dans le prolongement des travaux de la première auteure vers la réalisation d'outils de performance live en musique électronique de danse permettant, par exemple, de changer l'harmonie du morceau en temps-réel [**?**]. Ils se placent aussi dans le prolongement des travaux du second, l'utilisation du concept de graphe de scène et l'étude des modèles algébriques induits ayant déjà été initiées dans la thèse de doctorat de Simon Archipoff [**?**].

Plusieurs modélisations de la structure musicale utilisent déjà, sans le faire de manière explicite, une modélisation pouvant être rapprochée aux hiérarchies de transformations utilisées en graphisme. On trouve par exemple la possibilité de modéliser la macro structure d'un morceau musical à partir d'un arbre spécifiant les débuts et fins des différents motifs et sous-motifs[**?**]. Cependant, la partition et les éléments musicaux ne sont pas embarqués dans cette représentation.

En OpenMusic, la structure rythmique est modélisée en étiquetant les nœuds des arbres par des divisions du temps [**?**]. Bien qu'il soit possible de manipuler ces arbres avec des transformations, ces arbres ne permettent de ne représenter que le rythme. Il souffre aussi d'une représentation linéaire de l'écoulement du temps, sans doute largement héritée des travaux en théorie des langages formelles de Chomsky et appliqués à la musique via la notion de grammaires musicales [**?**]. Le modèle proposé ici, en autorisant une superposition partielle des structures musicales [**?**] s'affranchit de cette linéarité temporelle.

Au delà de la structure rythmique, OpenMusic permet également de représenter la structure hiérarchique grâce aux *maquettes*, développée comme module d'OpenMusic notamment sous l'impulsion du compositeur Tristan Murail. D'une certaine façon, le modèle proposé ici peut être vu comme une généralisation algébrique des maquettes d'OpenMusic.

En informatique graphique

Les *hiérarchies de transformations* [**?**] permettent déjà d'exprimer les liens d'articulations entre les différents éléments d'un squelette, en faisant en sorte qu'une transformation appliquée à un élément soit répercutée sur les éléments auxquels il est connecté. Les graphes de scènes peuvent ainsi fournir une représentation symbolique dont on peut obtenir un rendu en explicitant sa sémantique [**?**].

Ainsi, plusieurs langages de modélisation graphique utilisent explicitement les graphes de scène pour décrire symboliquement une scène, comme le langage X3D [**?**]. Dans un contexte temporisé, le langage SMIL [**?**] permet aussi de structurer et synchroniser du contenu multimédia. Cependant son utilisation des graphes de scène s'applique surtout à la partie graphique du langage : sa représentation temporelle repose sur un *graphe temporel* qui permet de grouper des éléments temporels, de les jouer en séquence ou en parallèle, mais ne permet pas l'application de transformations temporelles. Le logiciel Ossia [**?**], propose lui aussi une structure s'apparentant à un graphe temporel, permettant d'expliciter des scénarios multimedia avec lesquels on peut interagir en live.

2. UNE PREMIÈRE MODÉLISATION

Nous allons commencer par décrire comment une partition peut être vue comme une liste d'éléments atomiques (que nous appellerons dorénavant *atomes*) placés dans le temps. Dans le cas des partitions au sens traditionnel du terme, les éléments atomiques qui nous intéressent sont des notes.

Nous nous appliquerons en particulier à faire en sorte que notre modélisation ne dépende pas de la manière dont les notes et le temps sont modélisées. Cet abstraction est à la base de la flexibilité de notre langage de modélisation. Ainsi, notre langage peut être instantié sur une représentation des notes et du temps spécifiée en fonction du morceau que l'on souhaite représenter.

2.1. Temps et hauteurs dans Do ré mi, la perdrix

Pour représenter *Do ré mi, la perdrix*, dont la partition est donnée en introduction, nous allons nous contenter d'une modélisation minimaliste du temps et des hauteurs de note, adaptée spécifiquement à la représentation de cette comptine.

Durées : Un temps musical correspond à la durée écoulée entre deux pulsations : une noire dure 1 temps, une croche $\frac{1}{2}$ temps, une blanche 2 temps, etc. Nous représenterons donc les durées comme des fractions, autrement dit des nombres rationnels :

```
type Dur = Rational
```

Positions temporelles : Pour positionner les notes de manière absolue dans le temps, on se fixe une origine arbitraire 0 et on considère la durée écoulée depuis cette origine. Ainsi, la position temporelle 1 correspond à l'instant situé à un temps après l'instant 0, -2 à l'instant situé deux temps avant 0, et ainsi de suite.

```
type Time = Rational
```

Hauteurs de note : Les hauteurs de notes sont représentées de manière diatonique. On considère que 0 est le `do` du milieu, -1 est le `si` placé juste en dessous, 1 correspond à `ré`, 2 à `mi`, 8 au `do` à l'octave du dessus, et ainsi de suite.

```
type Pitch = Int
```

2.2. Positionnement absolu

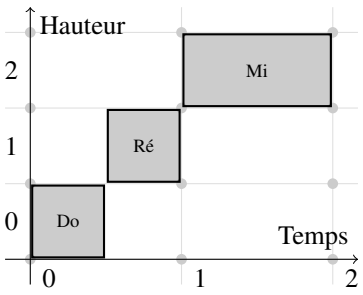
Nous allons à présent proposer une première syntaxe pour représenter *Do ré mi, la perdrix*. Pour placer un atome `a` à une position `x`, on utilise l'instruction `atom x a`. Dans notre exemple, les atomes sont des notes paramétrées par leur hauteur et leur durée. L'instruction `atom 1 (2,1)` correspond alors à un note commençant 1 temps après le début du morceau, cette note est un `mi` car sa hauteur est 2 et c'est une noire car sa durée est de 1. On peut alors représenter le motif « do ré mi » , c'est-à-dire la première mesure de *Do ré mi, la perdrix* :

```
drm1 :: HTile D Time (Pitch,Duration)
drm1 = do
  atom  0     (0,1/2) ;
  atom (1/2)  (1,1/2) ;
  atom  1     (2,1  )
```

Commençons par préciser que le code précédent est compilable en Haskell. Bien que l'implémentation de notre langage soit embarquée dans Haskell, la connaissance de ce langage n'est pas nécessaire à la lecture de l'article. Nous nous contenterons de souligner que les mots-clés de notre langage sont en gras et en vert et les mots-clés de Haskell en bleu et en italique. Le mot-clé `do` rend la syntaxe de notre langage possible au sein d'Haskell, et on peut l'ignorer complètement sans que cela nuise à la compréhension de cet article.

La première ligne indique à Haskell que nous sommes en train d'utiliser des tuiles hiérarchiques pour modéliser la structure du morceau. On explicite qu'on veut utiliser le type `Time` pour les positions temporelles et des paires composée d'une hauteur et d'une durée pour les atomes.

On définit ensuite le motif `drm1` comme une liste de notes positionnées dans le temps. On peut illustrer le contenu du motif `drm1` par un piano roll, qui rend compte à la fois de la position temporelle, de la hauteur des notes et de leur durée :



Bien que très élémentaire, ce premier programme met en évidence un aspect important de notre modèle : il est paramétré par le type des positions et le type des atomes. Bien que notre modélisation du temps et des notes soit extrêmement pauvre, elle suffit à l'étude de *Do ré mi, la perdrix*. Ainsi, plutôt que d'essayer de capturer toutes les subtilités de tous les genres musicaux existants dans une seule modélisation du temps ou une seule notion d'atome musical, nous préfèrerons ici laisser la personne utilisant notre modèle les spécifier. C'est précisement cette approche qui permet la flexibilité de notre modèle [2].

2.3. Sémantique formelle

Nous allons donner ici une sémantique formelle pour ce premier langage que nous venons de proposer, c'est-à-dire des instructions de la forme `atom x a` séparées par des point-virgules.

Soit X un ensemble de *positions*, que l'on appelle l'*espace*, et A un ensemble d'*atomes*. Un *motif simple* est un

2. au point que, en plus de la musique, on peut représenter de nombreux médias différents, comme par exemple les images, où l'espace n'est plus le temps mais le plan $\mathbb{R}^2$, et les atomes pourraient être par exemple des formes géométriques

mot sur les paires de position et d'atomes, c'est-à-dire un élément du monoïde libre $(X \times A)^*$. Autrement dit, un motif simple est une liste d'atomes placés dans l'espace. On notera ϵ le mot vide et $\cdot$ la concaténation. Les motifs simples sont donc des mots de la forme $[(x_1, a_1) \dots (x_n, a_n)]$.

Dans l'exemple de *Do ré mi, la perdrix*, l'espace $X = \mathbb{Q}$ est modélisé par les nombres rationnels et les atomes $A = \mathbb{Q} \times \mathbb{Z}$ sont les paires composées d'une durée et d'une hauteur de note diatonique.

On définit alors la sémantique dénotationnelle pour les motifs simples $[\![\_]\!]_s$ suivante :

$$[\![\texttt{atom x a}]\!]_s = [(\mathtt{x}, \mathtt{a})]$$
$$[\![m_1 \ ; \ m_2]\!]_s = [\![m_1]\!]_s \cdot [\![m_2]\!]_s$$

Ainsi la sémantique d'`atom x a` est un mot composé d'une sule paire (x, a) et celle du point-virgule est la concaténation.

3. ESPACE ET TRANSFORMATIONS

On aimerait à présent pouvoir transformer nos motifs construits par juxtaposition d'atomes. Pour cela, on applique une transformation à leur espace sous-jacent, ce qui a pour effet de déplacer les atomes.

Par exemple, une autre version de *Do ré mi, la perdrix* place le temps fort sur la note *mi* et considère les deux premières notes *do* et *ré* comme une anacrouse. En appliquant la translation $t \mapsto t-1$ qui décale toutes les positions d'un temps vers le passé, on obtient le motif correspondant à cette version en anacrouse représenté à la Figure **??**.

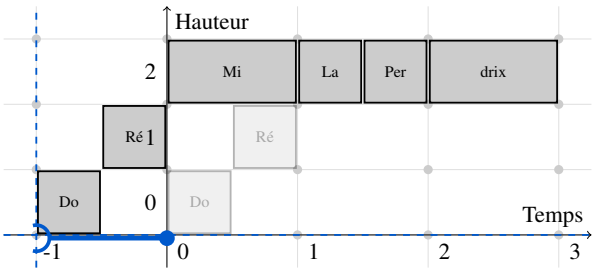


Figure 2. « Do ré mi, la perdrix » avec une anacrouse

3.1. Paramètre d'atome ou dimension de l'espace ?

Puisque nous pouvons déplacer les motifs dans le temps, nous aimerions aussi pouvoir les déplacer dans les hauteurs, c'est-à-dire les transposer diatoniquement. Ainsi, on peut obtenir les hauteurs de note du motif « mi fa sol, elle s'envole » en transposant diatoniquement le motif « do ré mi, la perdrix », plus précisément en ajoutant 2 à toutes les hauteurs de note, comme on peut le voir à la figure **??**.

Pour l'instant, les hauteurs de note sont un paramètre des atomes et ne font pas partie de l'espace. Puisque nous voulons les transformer, nous allons les considérer comme une dimension supplémentaire de notre espace, et donc comme une dimension supplémentaire de l'espace :

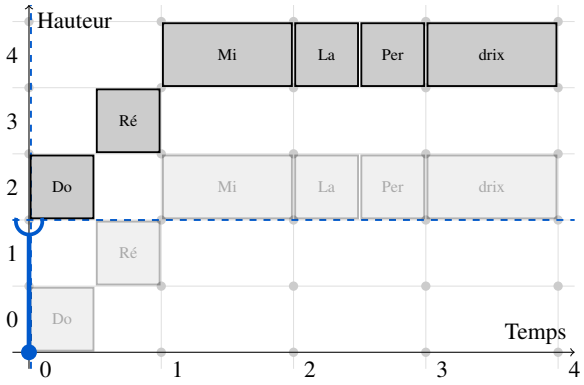


Figure 3. Transposition diatonique de 2 degrés du motif « do ré mi, la perdrix »

```
drm2 :: HTile D (Time,Pitch) Duration
drm2 = do
  atom ( 0 ,0) (1/2) ;
  atom (1/2,1) (1/2) ;
  atom ( 1 ,2)   1
```

De la même manière, on pourrait décider que les durées sont aussi une dimension de l'espace. On pourrait alors diviser toutes les durées par 2 pour obtenir un staccato (voir Figure **??**). Cependant, nous n'avons pas besoin de telles transformations dans cet article, nous laisserons donc les durées comme paramètre d'atome.

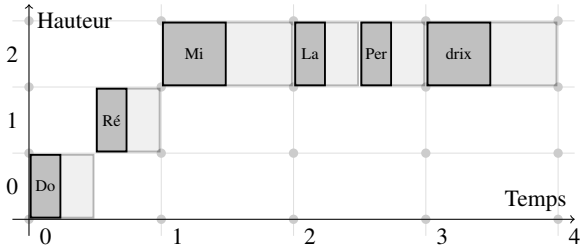


Figure 4. Division par 2 de la durée des notes

De manière plus générale, la différence entre dimension de l'espace et paramètre des atomes est avant tout une différence de fonction : le paramètre considéré a-t-il vocation à être transformé ou bien donne-t-il une simple information sur l'atome considéré ? Dans le premier cas, on préfèrera le voir comme une dimension de l'espace, dans le second comme un paramètre des atomes.

3.2. Les transformations dans *Do ré mi, la perdrix*

Pour notre modélisation structurelle de *Do ré mi, la perdrix*, nous avons choisi un espace à deux dimensions, à savoir les positions temporelles et les hauteurs de note, et des atomes qui n'ont qu'un seul paramètre : la durée de la note.

```
type Space = (Time,Pitch)
```

Pour garder la flexibilité de notre modélisation, nous devons spécifier à notre langage les transformations que

nous voulons utiliser, ici les fonctions de notre espace dans lui-même :

```
newtype Transf = Transf (Space -> Space)
```

Nous utiliserons en tout 5 types de transformations de l'espace, qui, en les composant, permettent de rendre compte similarités entre les différents motifs composant *Do ré mi la perdrix*. Pour chaque transformation, nous donnons le code Haskell qui permet de la spécifier à notre langage, ainsi que la fonction mathématique que ce code représente.

Identité La transformation identité, qui laisse inchangé un motif

```
idle = Transf id
```

Délai Les délais permettent de comparer deux motifs séparés dans le temps. Pour $d \in \mathbb{Q}$ un délai, ce sont des fonctions de la forme

$$\begin{aligned} \mathbb{Q} \times \mathbb{Z} &\to \mathbb{Q} \times \mathbb{Z} \\ (t, h) &\mapsto (t + d, h) \end{aligned}$$

où t et h sont respectivement les position temporelles et les hauteurs de note. On code cette fonction de la manière suivante :

```
del d = Transf (\(time,pitch) -> (time + d,
    pitch))
```

Transposition diatonique Comme nous l'avons vu, les transpositions diatoniques, de la forme $(t, h) \mapsto (t, h + i)$ avec $i \in \mathbb{Z}$ un intervalle diatonique, mettent en évidence la similarité entre « do ré mi, la perdrix » et « mi fa sol, elle s'envole »

```
transp i = Transf (\(time,pitch) -> (time,
    pitch + i))
```

Inversion On observe que les motifs « fa mi ré, dans un pré » et « mi ré do » près de l'eau sont le résultat de l'application d'une symétrie verticale au motif « do ré mi, la perdrix » (voir Figure **??**). Ainsi, pour obtenir le motif « fa mi ré, dans un pré », on commence par multiplier par -1 toutes les hauteurs des notes du motif « do ré mi, la perdrix » puis on les transpose diatoniquement d'une quarte. Autrement dit, on commence par appliquer l'inversion $(t, p) \mapsto (t, -p)$ puis la transposition $(t, p) \mapsto (t, p + 4)$.

```
inv = Transf (\(time,pitch) -> (time,-pitch))
```

Projection Le motif « la perdrix » se fait sur le même rythme que « do ré mi ». En fait, « la perdrix » correspond à la projection du motif « do ré mi » sur l'axe des hauteurs de note (voir Figure **??**). Autrement dit, on remplace toutes les hauteurs par 0, puis on transpose de 2 pour obtenir « la perdrix ».

```
proj = Transf (\(time,pitch) -> (time,0))
```

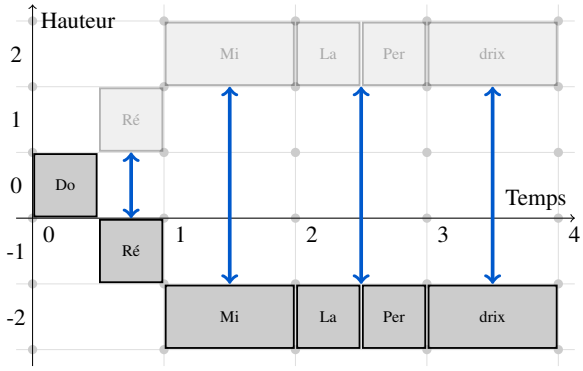


Figure 5. L'inversion $(t, p) \mapsto (t, -p)$

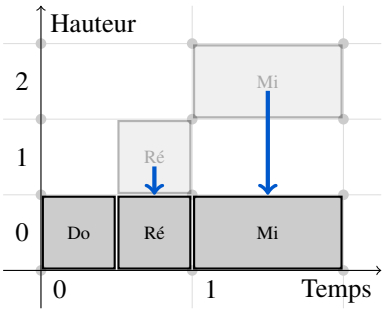


Figure 6. La projection $(t, p) \mapsto (t, 0)$

4. MODÉLISATION DE LA STRUCTURE

Nous avons mis en évidence la similarité de différents motifs de *Do ré mi, la perdrix* à l'aide de transformations de leur espace sous-jacent. Nous aimerions à présent expliciter sa structure à l'aide de ces similarités.

Pour cela, nous allons introdure les motifs hiérarchiques, un type de donnée proche des hiérarchies de transformations utilisées en graphisme. Nous introduirons de plus une syntaxe pour construire ces motifs. Enfin, nous donnerons une sémantique formelle à cette syntaxe.

4.1. Motifs hiérarchiques

Un *motif hiérarchique* est une liste d'arbres dont les nœuds sont étiquetés par des tranformations et les feuilles sont des atomes positionnés dans l'espace. La *profondeur* d'un motif hiérarchique est alors la plus grande profondeur parmis tous les arbres de cette liste.

L'intérêt de voir un motif hiérarchique comme une liste d'arbres, et non comme un simple arbre est qu'on a alors accès à la concaténation : nous pouvons continuer d'utiliser le point-virgule de la même manière que dans la section précédente.

Pour illustrer l'idée derrière les motifs hiérarchiques, allons à présent construire une représentation structurée complète de *Do ré mi, la perdrix*. Pour cela, nous aurons besoin au total de seulement 3 atomes :

```
c = atom (0  ,0) (1/2) -- la croche do
d = atom (1/2,1) (1/2) -- la croche ré
e = atom (1  ,2) (1/2) -- la noire mi
```

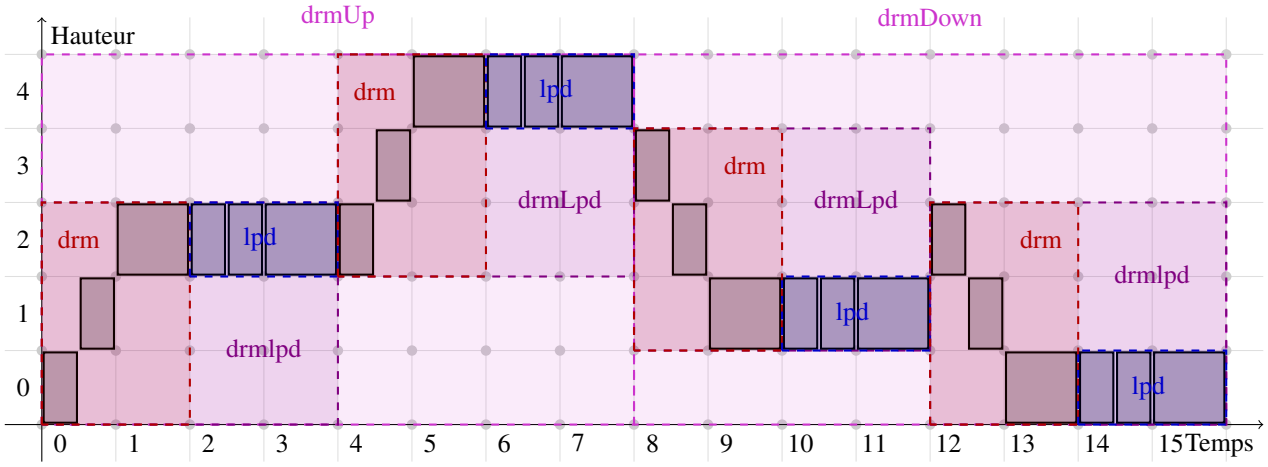


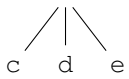
Figure 7. Une structuration complète de *Do ré mi, la perdrix* avec le programme `drmfull`

On peut alors, comme précédemment, représenter le motif « Do ré mi » :

```
drm = do
  c; d; e
```

En considérant `c`, `d` et `e` comme des feuilles, on peut voir `drm` comme une liste de 3 arbres : c'est un motif hiérarchique de profondeur 0.

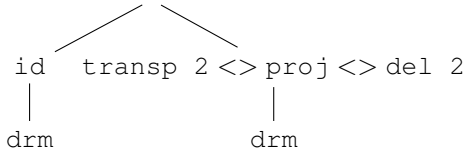
Visuellement, on représente un motif hiérarchique comme un arbre dont la racine n'est pas étiquetée. La profondeur du motif est alors la profondeur de cet arbre décrémentée de 1. Le motif hiérarchique `drm` est alors représenté par l'arbre suivant :



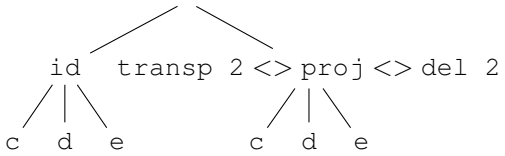
On peut alors construire le motif « Do ré mi, la perdrix » en combinant « Do ré mi » avec une version applatie et décalée de lui-même :

```
drmLpd = do
  idle                        |> drm ; -- do ré mi
  transp 2 <> proj <> del 2 |> drm -- la perdrix
```

Dans le code qui précède, l'opérateur `<>` représente la composition ∘ de transformations et l'instruction `d |> m` encapsule le motif `m` comme un sous-motif sur lequel agit la transformation `d`. Le code précédent correspond à l'arbre suivant :



En substituant `drm` par sa valeur, on obtient un motif hiérarchique de profondeur 1 représentant le motif « Do ré mi, la perdrix » :



Continuons de monter dans la hiérarchie : maintenant que nous avons construit « Do ré mi, la perdrix », nous savons que nous pouvons obtenir « mi fa sol, elle s'envole » en combinant délai et transposition. Nous pouvons alors définir le motif hiérarchique `drmUp`, qui représente la partie montante de *Do ré mi, la perdrix* :

```
drmUp = do
  -- do ré mi, la perdrix
  idle |> drmLpd ;
  -- mi fa sol, elle s'envole
  transp 2 <> del 4 |> drmLpd
```

De manière similaire, on peut obtenir la partie descendante, mais cette fois en appliquant des inversions :

```
drmDown = do
  -- fa mi ré, dans un pré
  transp 4 <> inv |> drmLpd ;
  -- mi ré do, près de l'eau
  transp 2 <> inv <> del 4 |> drmLpd
```

Une représentation complète de *Do ré mi, la perdrix* est alors donnée par la figure **??** et par le programme `drmFull` :

```
drmFull = do
  idle |> drmUp ;
  del 8 |> drmDown
```

4.2. Sémantique formelle

Nous avons enrichi notre langage de modélisation avec de nouvelles instructions de la forme `d |> m`. Nous allons à présent donner une sémantique hiérarchique $[\![\_]\!]_h$ de notre langage, qui correspond à l'intuition que nous en avons donné à la section précédente.

Soit D un ensemble de transformations. Pour définir formellement les motifs hiérarchiques, on définit inductivement l'ensemble $T(D, X, A)$ des arbres dont les noeuds

sont étiquetés par des transformations de D et les feuilles sont des paires de $X \times A$ composées d'une position et d'un atome :

1. toute paire (x, a) d'une position $x \in X$ et d'un atome $a \in A$ est un arbre de $T(D, X, A)$
2. toute paire (d, h) composée d'une transformation $d \in D$ et d'une liste $h \in T(D, X, A)^*$ de sous-arbres est aussi un arbre de $T(D, X, A)$.

Les motifs hiérarchiques sont alors des listes (ou encore des mots) de tels arbres, c'est-à-dire des éléments du monoïde libre $T(D, X, A)^*$. Ainsi, un arbre t est soit une feuille $t = (x, a)$ contenant une position $x \in X$ et un atome $a \in A$, soit un nœud $t = (d, h)$ contenant une transformation $d \in D$ et un motif hiérarchique $h \in T(D, X, A)^*$. Cette remarque justifie le fait de représenter visuellement un motif hiérarchique comme un arbre dont la racine n'est pas étiquetée.

La sémantique hiérarchique $[\![\_]\!]_h$ de notre langage de modélisation est alors :

$$[\![\texttt{atom x a}]\!]_h = [(\texttt{x}, \texttt{a})]$$
$$[\![\texttt{m1 ; m2}]\!]_h = [\![\texttt{t1}]\!]_h \cdot [\![\texttt{t2}]\!]_h$$
$$[\![\texttt{d |> m}]\!]_h = [(\texttt{d}, [\![m]\!])]_h$$

Ici, `atom x a` est la liste contenant un seul arbre qui est lui-même une feuille (x, a), `m1 ; m2` est la concaténation de listes et `d |> m` est une liste contenant un unique arbre dont la racine étiquetée par la transformation `d` n'a qu'un seul enfant : le motif `m`.

Notons que l'instruction `atom x a`, avec la séquence et les instruction de la forme `d |> m` permettent d'exprimer n'importe quel motif hiérarchique, excepté le motif vide : la concaténation permet de construire n'importe quelle liste non vide, et les instructions `atom x a` et `d |> m` correspondent respectivement aux feuilles et aux nœuds de l'arbre.

5. POSITIONNEMENT RELATIF

Nous avons à présent un modèle de la structure musicale de *Do ré mi, la perdrix* grâce aux motifs hiérarchiques, ainsi qu'une syntaxe permettant d'exprimer un tel modèle. L'instruction `atom x a` permet de positionner de manière absolue des atomes dans l'espace, le point-virgule de combiner deux motifs entre eux et l'instruction `|>` permet à la fois de créer un sous-motif et de le transformer.

En particulier, dans les exemples précédents, l'instruction `|>` nous sert à la fois à transformer les motifs, via les transformations `inv` et `proj` par exemple, mais aussi à positionner ces motifs, via `del` et `transp`.

Dans cette section, nous allons expliciter comment positionner les atomes et les sous-motifs de manière relative, c'est-à-dire les uns par rapport aux autres. Nous commencerons par expliciter comment utiliser les délais `del` pour obtenir une notation relative similaire au format MIDI. Nous généraliserons ensuite cette idée en introduisant la notion de tuile, qui permet le positionnement relatifs via un sous-ensemble de transformations : les changements de repère. Enfin, nous verrons comment combiner tuiles et motifs hiérarchiques pour expliciter la structure musicale tout en distinguant positionnement et transformations.

5.1. Positionnement absolu ou relatif ?

Le MIDI offre un exemple typique de positionnement relatif, où, pour chaque nouvel évènement MIDI, on indique le temps écoulé depuis le dernier évènement. Ainsi, une piste MIDI est de la forme

$$\langle \texttt{d1}, \texttt{e1} \rangle \langle \texttt{d2}, \texttt{e2} \rangle \ldots \langle \texttt{dn}, \texttt{en} \rangle$$

où `e1`, ... `en` sont des évènements MIDI et `d1`, ..., `dn` représentent le délai écoulé entre l'évènement précédent et l'évènement courant.

En considérant les évènements MIDI comme des atomes, on peut utiliser le positionnement absolu avec l'instruction `atom x a` pour représenter une piste MIDI, qui, elle utilise un positionnement relatif :

```
midiTrack = do
  atom d1 e1;
  atom (d1 + d2) e2;
  ...
  atom (d1 + d2 + ... + dn) en
```

Une manière de voir le positionnement relatif dans MIDI est de considérer que les notes sont placées à la position d'un curseur déplacé par les délais successif. Ce sont les positions successives de ce curseur qui apparaissent comme une somme de délais dans le code précédent.

Autrement dit, une manière de positionner les atomes de manière relative serait de déplacer un curseur dans l'espace et d'écrire les atomes à la position de ce curseur. C'est précisément ce que permettent les instruction de la forme `change (del d)`, où `del d` est la transformation définie précédemment :

```
midiTrack = do
  change (del d1);
  atom 0 e1;
  change (del d2);
  atom 0 e2;
  ...
  change (del dn);
  atom 0 en
```

On remarque alors que la position absolue devient inutile : les atomes sont tous placés à l'origine et positionnés par tous les délais qui précèdent.

5.2. Tuilage

On peut généraliser l'idée de curseur se déplaçant dans l'espace par l'idée d'un repère local que l'on déplacerait et que l'on transformerait. On peut alors voir les transformations de l'espace telles que `del d` comme des changements de repères. La composition de tous les changements de repère rencontrés jusqu'au point courant du calcul est

alors le repère courant dans lequel seront positionnés tous les éléments qui suivent.

Par exemple, en voyant `del d` et `transp t` comme des translations du repère courant, on peut écrire le motif « Do ré mi » de manière relative :

```
drm2 = do
  -- écriture d'une croche sur l'origine
  atom (0,0) (1/2); -- do
  -- déplacement du repère courant d'un demi-
      temps vers le futur
  change (del (1/2));
  -- déplacement du repère courant de 1 degré
      diatonique vers le haut
  change (transp 1)

  atom (0,0) (1/2);   -- croche à l'origine
  change (del (1/2)); -- 1/2 temps vers le futur
  change (transp 1);  -- un degré vers le haut

  atom (0,0) (1/2)   -- noire à l'origine
```

Le code précédent peut être factorisé. En effet, après avoir dessiné une croche à l'origine, on a souvent envie de déplacer le repère d'un demi-temps vers le futur. Pour cela, on définit une *tuile*, c'est-à-dire un motif suivi d'un changement de repère. Par exemple, on peut définir les tuiles `quaverT` et `crotchetT` :

```
quaverT = do
  atom (0,0) (1/2);
  change (del (1/2))

crotchetT = do
  atom (0,0) 1;
  change (del 1)
```

La tuile `quaverT` permet d'écrire une note de durée `1/2` placée à l'origine du repère courant, puis elle déplace ce repère d'un demi-temps vers le futur. La tuile `crotchetT` fait de même, mais avec un temps complet au lieu d'un demi-temps. La Figure **??** représente visuellement une tuile en dessinant le motif qu'elle contient et le changement de repère qui a lieu après l'écriture de ce motif. Dans la figure, on note •—c les translations, où la boule • représente l'origine du repère avant sa translation et la douille c l'origine après la translation.

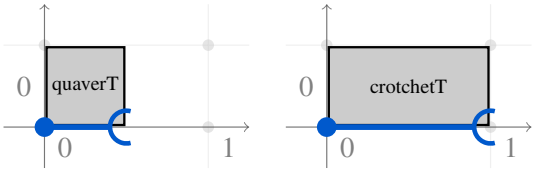


Figure 8. À gauche la tuile `quaverT` et à droite la tuile `crotchetT`

Une tuile peut aussi ne contenir qu'un changement de repère et pas de motif :

```
transpT i = do
  change (transp i)

delT i = do
  change (del i)
```

On peut alors écrire la tuile `drmT` contenant le motif « Do ré mi » à l'aide de la syntaxe tuilée :

```
drmT = do
  quaverT ;
  transpT 1 ;
  quaverT ;
  transpT 1 ;
  crotchetT
```

Notons que `drmT` est bien une tuile, et non un simple motif hiérarchique comme l'était `drm`. En effet, `drmT` contient d'une part le motif « Do ré mi » et d'autre part l'accumulation de toutes les transformations. La Figure **??** illustre la construction de la tuile `drmT`.

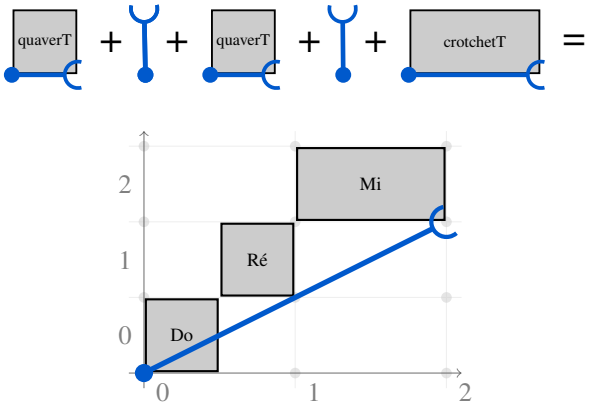


Figure 9. Construction de la tuile `drmT` à partir d'une somme de tuiles

On peut alors combiner la tuile `drmT` et une tuile `lpdT` contenant le motif « la perdrix » pour obtenir une tuile `drmLpdT` contenant le motif « Do ré mi, la perdrix » :

```
lpdT = do
  quaverT ;
  quaverT ;
  crotchetT

drmLpdT = do
  drmT ;
  lpdT
```

Ne nous arrêtons pas en si bon chemin ! On peut obtenir les 4 premières mesures de *Do ré mi, la perdrix*, c'est-à-dire la moitié montante de la comptine, en combinant la tuile `drmLpdT` avec elle-même :

```
drmUpT = do
  drmLpdT;
  drmLpdT
```

La Figure **??** illustre la construction de la tuile `drmUpT` à partir de `drmT`, `lpdT` et `drmLpdT`.

5.3. Tuilage hiérarchique

Nous venons de voir que, même sans utiliser l'instruction `|>`, la structure devient apparente dans le code du programme.

Pour autant, la représentation tuilée `drmUpT` présente deux inconvénients :

1. Nous n'avons pas pu utiliser la projection. En effet, c'est une fonction non bijective et l'appliquer comme un changement de repère aurait pour conséquence l'applatissement de tout ce qui suit.

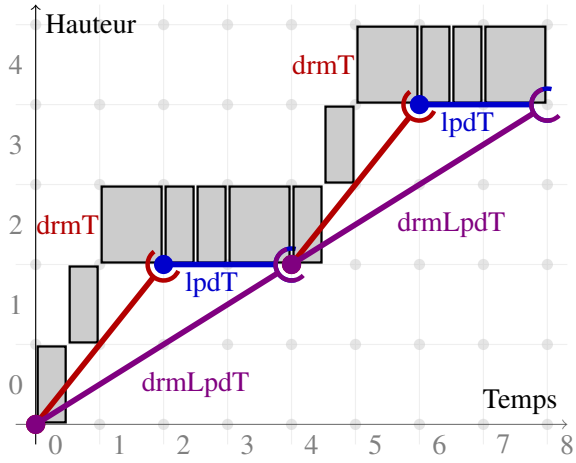


Figure 10. Construction de la tuile `drmUpT` à partir de `drmT` et `lpdT`

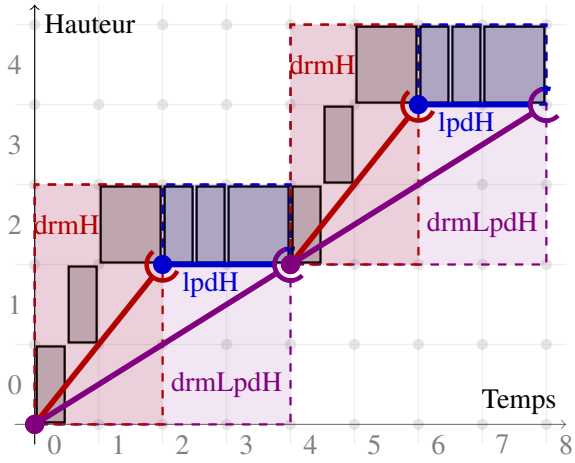


Figure 11. Construction de la tuile hiérarchique `drmUpH` à partir de `drmH` et `lpdH`

2. La structure n'est visible que dans le programme, mais pas dans sa sémantique, car nous n'avons pas explicité les niveaux hiérarchiques avec des instructions de la forme `d |> m`.

Nous allons donc combiner la notion de tuile avec celle de hiérarchie, en explicitant les changements de repère aux niveaux hiérarchiques supérieurs :

```
-- Do ré mi
drmH= do
  idle |> drmT;
  transpT 2;
  delT 2

-- la perdrix
lpdH = do
  proj |> drmT;
  delT 2

-- Do ré mi, la perdrix
drmLpdH = do
  idle |> do {
    drmH ; lpdH
  };
  transpT 2;
  delT 4
```

Dans le code précédent, en plus d'expliciter la hiérarchie, nous exprimons les positionnement relatifs des différents sous-motifs les uns par rapports aux autres. On peut alors écrire la tuile `drmUpH`, dont la construction est illustrée par la figure **??**, comme étant le tuile `drmLpdH` sommée avec elle :

```
-- Partie montante
drmUpH = do
  idle |> do {
    drmLpdH ; drmLpdH
  };
  transpT 4;
  delT 8
```

5.4. Sémantique formelle

Dans la syntaxe que nous venons présenter, nous avons la possibilité de terminer sur un changement de repère, qui sera ensuite appliquée à ce qui suivra. Pour pouvoir l'exprimer dans notre sémantique, nous allons introduire la notion de tuile. Il s'agit d'une paire composée d'un motif et d'une transformation. Lorsque le motif est simple, on parle de *tuile simple* et lorsqu'il est hiérarchique, on parle de tuile *hiérarchique*.

Pour combiner deux tuiles, on concatène le premier motif avec une version transformée du second motif. C'est cette combinaison de tuile qui nous servira de sémantique pour le point-virgule.

Formellement, supposons que l'ensemble D des transformations précédemment défini est muni d'une opération de composition associative $\circ$. On note alors id l'élément neutre de ce monoïde. Supposons de plus que D agit sur l'espace X via une action à gauche $\triangleright$. Pour être une action, $\triangleright$ doit satisfaire l'équation suivante :

$$(d_1 \circ d_2) \triangleright x = d_1 \triangleright d_2 \triangleright x \tag{1}$$

Tout élément $d \in D$ peut alors être vu comme une fonction $x \mapsto d \triangleright x$ de X dans lui-même, ce qui justifie le fait d'appeler les éléments de D des transformations.

On peut étendre l'action $\triangleright$ à une action de D sur les arbres $t \in T(D, X, A)$ en posant :

$$d \triangleright t = \begin{cases} (d \triangleright x, a) & \text{si } t = (x, a) \text{ est une feuille} \\ (d \circ d', h) & \text{si } t = (d', h) \text{ est un nœud} \end{cases}$$

On montre alors aisément que $\triangleright$ satisfait toujours (**??**) et est donc que D agit bien sur les arbres.

On peut alors étendre à nouveau l'action $\triangleright$ aux motifs hiérarchiques en posant

$$d \triangleright [t_1 \dots t_n] = [(d \triangleright t_1) \dots (d \triangleright t_n)]$$

En plus de continuer de satisfaire l'équation (**??**), l'extension point à point de l'action $\triangleright$ aux motifs hiérarchiques a la propriété supplémentaire suivante : appliquer une même transformation à deux motifs et concaténer les résultat re-

vient au même qu'appliquer cette transformation à la concaténation des deux motifs :

$$d \triangleright (h_1 \cdot h_2) = (d \triangleright h_1) \cdot (d \triangleright h_2) \qquad (2)$$

Les actions satisfaisant la condition **??** sont appelées des actions par morphismes.

Nous avons à présent tous les ingrédients pour opérer un produit semi-direct : deux semi-groupes et une action par morphismes d'un des semi-groupes sur l'autre. Une *tuile hiérarchique* est une paire $\langle h, d\rangle \in T(D, X, A)^* \rtimes D$ composée d'un motif hiérarchique h suivi d'un changement de repère d. La somme de tuiles est alors donnée par l'opération du produit semi-direct :

$$\langle h_1, d_1\rangle + \langle h_2, d_2\rangle = \langle h_1 \cdot (d_1 \triangleright h_2), d_2 \circ d_1\rangle$$

Cette somme tuilée correspond bien à l'intuition que nous en avons donné dans les sections précédentes : la transformation d_1, située après le motif h_1, agit sur le motif h_2, et agira également sur le motifs suivant car elle est composée avec d_2 pour agir sur les tuiles futures.

Grâce aux tuiles, nous allons pouvoir donner une sémantique à `change c`. Nous appelons $[\![\_]\!]_t$ la sémantique par tuiles hiérarchiques :

$$\begin{aligned}
[\![\texttt{atom x a}]\!]_t &= \langle([\texttt{x}, \texttt{a})], \mathrm{id}\rangle \\
[\![\texttt{change c}]\!]_t &= \langle \epsilon, \texttt{c}\rangle \\
[\![\texttt{t1 ; t2}]\!]_t &= [\![\texttt{t1}]\!]_t + [\![\texttt{t2}]\!]_t \\
[\![\texttt{d |> t}]\!]_t &= \langle[(\texttt{d}, h)], \mathrm{id}\rangle \qquad \text{où } \langle h, d'\rangle = [\![t]\!]_t
\end{aligned}$$

Dans cette sémantique, `atom x a` est la tuile dont le motif sous-jacent contient l'unique feuille $(\texttt{x}, \texttt{a})$ et la transformation effectuée après le motif est l'identité. `change c` est la tuile dont le motif sous-jacent est vide et la transformation est `c`. Le point-virgule a pour sémantique la somme tuilée. L'opérateur `d |> t` encapsule le motif hiérarchique h résultant de t et oublie le changement de repère d' en le remplaçant par la transformation identité.

6. CONCLUSION

Dans cet article, nous avons présenté un modèle algébrique pouvant servir de sémantique formelle à un langage de modélisation. Bien que l'exemple de *Do ré mi, la perdrix* soit rudimentaire, tous les éléments sont en place pour placer la partition dans des espaces plus riches et plus expressifs que l'espace diatonique utilisé ici.

Le langage que nous venons d'introduire a notamment pour but de rendre possible l'application de transformations, qu'elles soient rythmiques, mélodiques ou harmoniques en live à des motifs. Pour cela, la partition pourra être représentée par des motifs hiérarchiques et des transformations pourront être appliquées aux différents nœuds de la hiérarchie, permettant ainsi de manipuler la partition en temps-réel.

7. REFERENCES

[1] Archipoff, S. "Développement de la programmation par tuilage pour les systèmes multimédias interactifs", *Thèse de doctorat*, 2020.

[2] Barbar, K., Desainte-Catherine, M., Miniussi, A. "The Semantics of Musical Hierarchies", *Computer Music Journal*, pp. 30-37, 1993.

[3] Brutzman, D., Daly, L. "X3D : extensible 3D graphics for Web authors", *Elsevier*, 2010.

[4] Bulterman, D. et al. "Synchronized Multimedia Integration Language (SMIL 3.0)", *Recommandation W3C*, www.w3.org/TR/2008/REC-SMIL3-20081201/smil-timing.html, 2008.

[5] Calandra, J., Chouvel, J.-M., Desainte-Catherine, M., Michel, E. "Visualisation of Multi-scale Formal Diagrams for Music Analysis", *2023 4th International Symposium on the Internet of Sounds*, pp. 1-9, 2023.

[6] Celerier, J.-M. et al. "OSSIA : Towards a unified interface for scoring time and interaction", *TENOR 2015-First International Conference on Technologies for Music Notation and Representation*, 2015.

[7] Cope, D. "Hidden structure : music analysis using computers", *AR Editions, Inc.*, 2009.

[8] Guichaoua, C., Bimbot, F. "Inférence de segmentation structurelle par compression via des relations multi-échelles dans les séquences d'accords", *JIM 2018-Journées d'Informatique Musicale*, 2018.

[9] Hiller, L. "Computer music", *Scientific American*, 1959.

[10] Jacquemard, F., Donat-Bouillud, P., Bresson, J. "A structural theory of rhythm notation based on tree representations and term rewriting", *International Conference on Mathematics and Computation in Music*, 2015.

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

[12] Lerdahl, F., Jackendoff, R. "A generative theory of tonal music", *MIT Press*, 1983.

[13] Mesnage, M., Riotte, A. "Modélisation informatique de partitions, analyse et composition assistée", *Cahiers de l'Ircam nř 3, Recherche musicale : La composition assistée par ordinateur*, 1993.

[14] Mukundan, R. "Advanced methods in computer graphics : with examples in OpenGL", *Springer Science & Business Media*, 2012.

[15] Rixte, A. "LiveScaler : Contrôler en live l'harmonie d'un morceau de musique électronique", *Journées d'Informatique Musicale (JIM)*, 2023.

[16] Tobler, R. "Separating semantics from rendering : a scene graph based architecture for graphics applications" *The Visual Computer*, 27, pp. 687-695, 2011.

[17] Zimmermann, D. "Modelling Musical Structures", *Constraints*, 6, pp. 53-83, 2001.