PROGRAMMER AVEC DES TUILES MUSICALES: LE T-CALCUL EN EUTERPEA

Paul Hudak
Department of Computer Science
Yale University
New Haven, CT 06520-8285
paul.hudak@yale.edu

David Janin
LaBRI, CNRS UMR 5800,
IPB, Université de Bordeaux,
F-33405 Talence
david.janin@labri.fr

Résumé

Euterpea est un langage de programmation dédié à la création et à la manipulation de contenus media temporisés – son, musique, animations, vidéo, etc.... Il est enchassé dans un langage de programmation fonctionnelle avec typage polymorphe : Haskell. Il hérite ainsi de toute la souplesse et la robustesse d'un langage de programmation moderne.

Le T-calcul est une proposition abstraite de modélisation temporelle qui, à travers une seule opération de composition : le produit tuilé, permet tout à la fois la composition séquentielle et la composition parallèle de contenus temporisés.

En présentant ici une intégration du T-calcul dans le language Euterpea, nous réalisons un outil qui devrait permettre d'évaluer la puissance métaphorique du tuilage temporel combinée avec la puissance programmatique du langage Euterpea.

1. INTRODUCTION

Programmer des pièces musicales.

De nos jours, il existe de nombreux langages de programmation dédiés à la composition musicale tels que, parmi d'autres, le langage Euterpea en Haskell [10] qui s'appuie sur la bibliothèque Haskore [13], le langage Elody [25] qui étend le λ-calcul, ou encore CLISP/ CLOS [4] pour OpenMusic [1].

Ces langages permettent de décrire des séquences de notes ou des agencements de flux audio. Ils offrent aussi, de plus en plus, des opérations abstraites de manipulation de séquences musicales entières, en intégrant des concepts de programmation classiques. La musique écrite dans ces langages peut ainsi résulter de l'*exécution* d'algorithmes de génération de flux musicaux.

Dans ces langages, bon nombre de structures de données et de contrôles usuelles des langages de programmation sont ainsi disponibles pour cette écriture algorithmique. Elles permettent de prendre de la hauteur en offrant aux compositeurs des métaphores de modélisation musicale plus adaptées à une pensée musicale abstraite. Paradoxalement, les structures musicales induites par ces langages de programmation peuvent aussi souffrir de limitations.

Les concepts de listes, d'arbres ou même de fonctions, sont bien entendu applicables à une écriture algorithmique de la musique. Les travaux autours de la linguistique appliquée à la musicologie [24] témoigne de leur pertinence. Cependant, de nombreuses constructions musicales ne s'y prêtent pas vraiment comme l'illustre, par exemple, des pièces polyrythmiques à la structure complexe telles que les arabesques de Debussy.

Programmer l'espace et le temps.

Une problématique majeure à laquelle ces langages doivent répondre est de rendre compte des caractéristiques spatiales et temporelles du langage musical.

Dans une approche classique, la *dimension temporelle* est modélisée par le *produit séquentiel* : la concaténation de deux listes, et la *dimension spatiale* est modélisée par le *produit parallèle*. Cette approche est formalisée dans [9]. Elle conduit à définir l'algèbre des flux média polymorphes. Applicable à la musique comme dans le *Domain Specific Language* Euterpea[10] basé sur la librairie Haskore [13] réalisé en Haskell, elle est aussi applicable à tout flux media temporisé tels que les flux vidéo, les animations ou même les flux de contrôle-commande [8].

Une étude récente des structures rythmiques [15] montre cependant que la distinction faite entre composition séquentielle et parallèle n'est pas compatible avec une description hiérarchique, multi-échelle, des structures musicales. Par exemple, un départ en anacrouse peut empiéter sur le flux musical qui le précède. Il induit un parallélisme local. Pourtant, au niveau de l'intention musicale, plus abstraite, il pourra apparaitre lors de la composition séquentielle de deux mélodies.

Bien sur, ces superpositions locales pourraient être traitées comme des exceptions à cette distinction séquentielle/parallèle. Notre proposition consiste, au contraire, à les expliciter. Ce faisant, la composition n'est plus séquentielle ou parallèle : elle est tuilée.

travail partiellement soutenu par ANR-12-CORD-0009

La modélisation par tuilage spatio-temporel.

Dans la continuité des propositions existantes pour la programmation musicale [1, 25, 10], comme dans la continuité des approches en musicologie formelle [24], la *modélisation par tuilage* permet d'unifier les notions de compositions séquentielles et parallèles.

Plus précisément, en nous appuyant sur le concept de *flux media temporisés* [8, 9] enrichi par des *marqueurs de synchronisation* [3], nous obtenons une notion de *flux media temporels tuilés*. Le produit associé, appelé *produit tuilé*, apparaît tout à la fois comme un opérateur séquentiel et un opérateur parallèle.

En pratique, la composition tuilée s'inspire de travaux déjà anciens. Le produit de tuilage ne fait qu'offrir une alternative à la notion de barre de mesure en musique. Il permet par exemple de modéliser la notion musicale d'anacrouse [15]. Le produit tuilé apparaît déjà implicitement dans le langage LOCO [6], une adaptation du langage *Logo* à la manipulation de flux musicaux.

La formalisation du produit tuilé, nous a conduit à proposer une algèbre dédiée à la synchronisation des flux musicaux [3]. Deux outils de manipulations de flux audio tuilés, la librairie *libTuiles* et l'interface *liveTuiles* [22], dérivent de cette approche. In fine, une proposition abstraite : le T-Calcul [21], propose d'intégrer ces concepts à un langage de programmation.

En théorie, le produit de tuilage apparaît aussi dans les monoïdes inversifs [23]. Son usage comme outil de modélisation pour les systèmes informatisés semble prometteur [20]. Le produit tuilé, tel que nous proposons de le manipuler en vue d'applications musicales, est donc une construction particulièrement bien fondée au cœur d'une théorie mathématique dont la robustesse n'est plus à démontrer.

2. TUILER LES FLUX TEMPORISÉS

Nous décrivons ici comment les flux media tuilés peuvent être construit à partir des flux media en les enrichissant simplement de deux marqueurs de synchronisation. Cette construction peut être vue comme une abstraction des notions de synchronisation et de mixage telle qu'elle sont couramment mise en œuvre dans les langages de programmation musicale comme dans les studios de montage audio.

2.1. Flux media temporisés

Un *flux média temporisé* [8] est une structure abstraite représentant des données qui sont positionnées dans le temps relativement au début du flux media. Cette notion est illustrée Figure 1 sur un axe temporel, implicite, allant du passé, à gauche, vers le futur, à droite. Parfois produites par des programmes, ces structures peuvent être de durée infinie.

Figure 1. Un flux média temporisé fini m_1 et un flux média temporisé infini m_2.

Les opérations naturellement associées à ces flux média temporisés sont : la *composition séquentielle*

$$m_1 :+: m_2$$

qui modélise le lancement de deux flux média temporisés l'un après l'autre, le premier étant nécessairement fini, et, la *composition parallèle*

$$m_1 :=: m_2$$

qui modélise le lancement de deux flux média temporisés en même temps. Ces notions sont illustrés Figure 2.

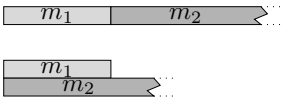

Figure 2. Composition séquentielle $m_1 :+: m_2$ et composition parallèle $m_1 :=: m_2$ des flux m_1 et m_2.

Il apparait que ces deux opérations sont des cas particuliers d'une opération plus générale : le *produit synchronisé* ou *produit tuilé*.

2.2. Flux media temporisés tuilés

Un flux media temporisé tuilé est définit comme un flux media temporisé m enrichi de deux marqueurs de synchronisation *pre* et *post* qui sont définis par la distance, mesurée en temps, qui les sépare du début du flux. Autrement dit, un flux media tuilé t peut simplement être codé comme un triplet

$$t = (pre, post, m)$$

avec *pre* et *post* définit comme deux rationnels positifs et m un flux media. Un tel flux tuilé est illustré Figure 3.

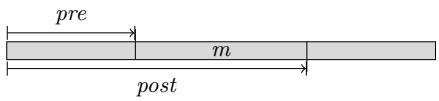

Figure 3. Un flux media temporel.

Remarque. Cette définition des flux tuilés reste valide avec des flux infinis, la position de référence pour positionner *pre* et *post* sur un flux étant toujours le début de ce flux.

2.3. Produit tuilé

Le *produit tuilé* $t_1 \,\%\, t_2$ de deux flux tuilés de la forme $t_1 = (pre_1, post_1, m_1)$ et $t_2 = (pre_2, post_2, m_2)$,

se définit alors en deux étapes successives. La *synchronisation*, qui consiste à positionner les deux flux t_1 et t_2 l'un par rapport à l'autre, de telle sorte que le marqueur de sortie $post_1$ du premier flux tuilé t_1 coïncide avec le marqueur d'entrée pre_2 du second flux tuilé t_2. La *fusion*, qui consiste alors à fusionner [1] les deux flux sous-jacents ainsi positionnés, en conservant pre_1 comme marqueur d'entrée et $post_2$ comme marqueur de sortie résultant de ce produit.

Cette construction est illustrée Figure 4. Dans cette

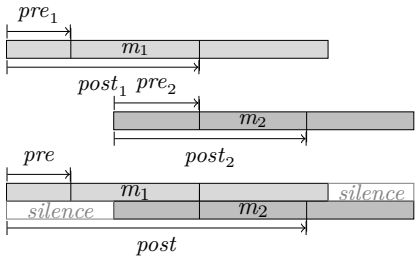

Figure 4. Une instance de produit synchronisé

figure le début du flux musical résultant provient de la première tuile. C'est là un cas particulier comme on peut le vérifier sur un autre example, décrit Figure 5.

3. IMPLÉMENTATION EN HASKELL

L'implémentation en *Haskell/Euterpea* des flux media tuilés et du produit tuilé ne fait que mettre en œuvre les schémas ci-dessus. Les opérateurs qui peuvent dériver de ces structures sont décrits par la suite, à la fois par leur implémentation en *Haskell* et par le schéma correspondant.

3.1. Le type tuile

Le type de donnée *Tile a* est codé de la façon suivante à partir du type *Music a*, qui est une instance particulière du type *Temporal a* :

```
data Tile a = Tile { preT  :: Dur,
                     postT :: Dur,
                     musT  :: Music a }
```

avec **type** *Dur* = *Rational*. On utilise ici le type *Rational* afin d'éviter les erreurs d'approximation qui pourraient résulter, par exemple, de l'utilisation du type *Float*.

Une fonction *durT* permet de calculer la *durée de synchronisation* d'un flux tuilé, c'est à dire, le temps relatif entre la marque de synchronisation *pre* et la marque de synchronisation *post*.

1 . Fusion qui dépendra du media considéré : mixage pour de l'audio, union pour de la musique, superposition pour de la vidéo, etc.

```
durT :: Tile a → Dur
durT (Tile pr po m) = po − pr
```

Dans le cas où le marqueur *pre* se trouve *avant* le marqueur *post* on dit que le flux tuilé est *positif*. Dans le cas contraire, lorsque le marqueur *pre* est situé *après* le marqueur *post*, on dit que le flux tuilé est *négatif*. Dans tous les cas, le flux media est inchangé. Il sera toujours produit du passé (représenté à gauche) vers le futur (représenté à droite).

3.2. Le produit tuilé

Le produit synchronisé, noté % est alors codé par :

```
(%) :: Tile a → Tile a → Tile a
Tile pr1 po1 m1 % Tile pr2 po2 m2 =
  let d = po1 − pr2
  in Tile (max pr1 (pr1 − d))
          (max po2 (po2 + d))
          (if d > 0 then m1 :=: mDelay d m2
           else mDelay (−d) m1 :=: m2)
```

où *mDelay* est la fonction définit par :

```
mDelay d m = case signum d of
   1 → rest d :+: m
   0 → m
 − 1 → m :+: rest (−d)
```

Dans ce codage, la fonction *mDelay* assure la phase de synchronisation, le produit parallèle :=: assure la phase de fusion. Implicitement, du silence est inséré de part et d'autre des flux musicaux sous-jaçent afin de permettre cette composition parallèle.

Selon la position des marqueurs de synchronisation, le début de la musique peut provenir de la première tuile, comme dans le cas de la Figure 4 ou bien de la seconde tuile, comme dans le cas de la Figure 5.

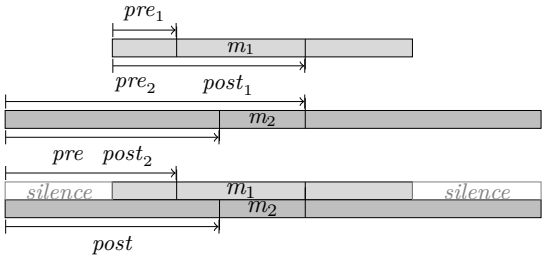

Figure 5. Une autre instance de produit synchronisé

3.3. Reset, co-reset et inverses

Trois fonctions sur les tuiles : le *reset*, le *co-reset* et l'*inverse*, découlent de ces définitions. Elles sont,

comme on le verra, liées les unes aux autres.

```
re, co, inv :: Tile a → Tile a
re  (Tile pre post m) = Tile pre pre m
co  (Tile pre post m) = Tile post post m
inv (Tile pre post m) = Tile post pre m
```

Leur effet sur une tuile t est décrit Figure 6.

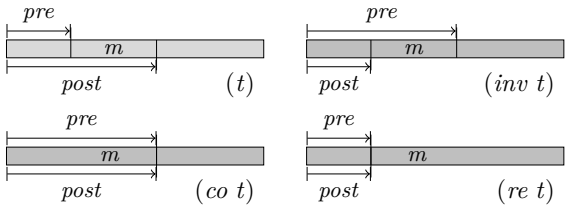

Figure 6. Reset, co-reset and inverse

3.4. Codage des tuiles musicales

Chaque note, de type *Music Pitch* en Euterpea, est décrite comme un triplet (n, o, d) avec un nom [2] n, une octave [3] o et une durée [4] d. Ces notes peuvent être converties en notes tuilées à l'aide de la fonction t définie par :

```
t :: (Octave → Dur → Music Pitch)
          → Octave → Dur → Tile Pitch
t n o d = if d < 0 then Tile d 0 (n o (−d))
          else Tile 0 d (n o d)
```

De même pour les silences, avec la fonction r définie par :

```
r :: Dur → Tile a
r d = if d < 0 then Tile d 0 (rest (−d))
      else Tile 0 d (rest d)
```

Pour jouer une tuile, on ne convient de ne garder que la partie musicale situé entre les marqueurs *pre* et *post*. Cette *projection* est réalisée par la fonction *tToM* décrite ci-dessous.

```
tToM :: Tile a → Music a
tToM (Tile pr po m) = takeM (po − pr)
                            (dropM pr m)
```

où les fonctions Euterpea *takeM* et *dropM* sur les objets de type *Music* permettent de supprimer ou extraire, comme sur les listes, des sous-séquences musicales selon le paramètre de durée passé en argument.

La musique codée dans une tuile peut alors être jouée par composition :

```
playT = play ∘ tToM
```

où la fonction *play* est la fonction en Euterpea permettant de jouer des séquences musicales de type *Music*.

3.5. Premier exemple

On peut ainsi proposer un premier exemple :

```
fj1 = t c 4 en % t d 4 en % t e 4 en % t c 4 en
fj2 = t e 4 en % t f 4 en % t g 4 qn
fj3 = t g 4 sn % t a 4 sn % t g 4 sn % t f 4 sn
    % t e 4 en % t c 4 en
fj4 = t c 4 en % t g 3 en % t c 4 qn
```

qui code le canon *Frère Jacques*. Il pourra être joué par la commande
```
test1 = playT (fjr % r 4)
```

Une fonction *tempoT* permet aussi de changer le tempo des tuiles. Elle est codée de la façon suivante, héritant en quelque sorte de la fonction *tempo* sur les objets musicaux sous-jacents :

```
tempoT :: Dur → Tile a → Tile a
tempoT r (Tile pr po m) =
  assert (r > 0) (Tile (pr/r) (po/r) (tempo r m))
```

On utilise ici la fonction *Control.Exception.assert* qui permet de vérifier que le coefficient r de changement de tempo est strictement positif.

Plus généralement, toute fonction agissant sur les objets musicaux peut être appliquée aux tuiles, lorsqu'on ne souhaite pas modifier les paramètres de synchronisation, grâce à la fonction d'ordre supérieur *liftT* définie par :

```
liftT :: (Music a → Music b) → (Tile a → Tile b)
liftT f (Tile pr po m) = Tile pr po (f m)
```

Ainsi, l'exemple suivant nous permettra de jouer *Frère Jacques* sur un piano Rhodes deux fois plus vite.
```
fjR = litfT (intrument rhodesPiano) fjr
test2 = playT (tempoT 2 (fjR % r 4))
```

4. ÉQUIVALENCE OBSERVATIONNELLE

Un concept important en Euterpea est l'équivalence observationnelle. Elle se généralise sur les flux musicaux tuilés nous permettant de donner un sens à l'affirmation « les flux tuilés t_1 et t_2 se comportent de la même façon ».

2 . En notation anglaise, de *a* pour *la* à *g* pour *sol*, avec *c* pour *do*, *cf* (*c flat*) pour *do* bémol, *cs* (*c flat*) pour *do* dièse, etc.

3 . C'est-à-dire un numéro d'octave, le do à la clé correspondant à *c* 4 ; il est précédé de *b* 3 et suivie de *d* 4, un clavier de piano allant typiquement de *a* 0, le *la* le plus grave, à *c* 8, le *do* le plus aigu.

4 . En valeur symbolique valant fraction de ronde, avec des constantes prédéfinies, en notation anglaise, telle que *wn* (*whole note*) pour la ronde, valant 1, ou bien *qn* (*quarter note*) pour la noire, valant 1/4, ou encore *en* (*eight note*) pour la croche, calant 1/8, etc.

4.1. Equivalence de flux

Deux flux musicaux m_1 et m_2 sont observationellement équivalent, ce qu'on note

$$m_1 \text{ `}equiv\text{` } m_2$$

lorsqu'ils produisent le même effet à l'execution, c'est à dire, lorsque *play* m_1 et *play* m_2 produisent le même effet, où *play* désigne la fonction *jouant* les flux musicaux.

Les détail sur cette équivalence *equiv* peuvent être trouvé dans [8, 9]. Pour ce qui nous interesse, il suffit de savoir qu'elle satisfait les axiomes suivants :

$$\begin{array}{rcl}
(m_1 :+: m_2) :+: m_3 & \text{`}equiv\text{`} & m_1 :+: (m_2 :+: m_3) \\
(m_1 :=: m_2) :=: m_3 & \text{`}equiv\text{`} & m_1 :=: (m_2 :=: m_3) \\
m_1 :=: m_2 & \text{`}equiv\text{`} & m_2 :=: m_1 \\
rest\ 0 :+: m & \text{`}equiv\text{`} & m \\
m :+: rest\ 0 & \text{`}equiv\text{`} & m
\end{array}$$

et, lorsque $dur\ m_1 = dur\ m_3$

$$\begin{array}{l}
(m_1 :+: m_2) :=: (m_3 :+: m_4) \\
\qquad \text{`}equiv\text{`}\ (m_1 :=: m_3) :+: (m_2 :=: m_4),
\end{array}$$

On suppose en outre que l'équation suivante

$$m \text{ `}equiv\text{` } (m :=: m)$$

est satisfaite pour tout flux tuilé m.

Remarque. Bien que raisonnable, cette dernière équivalence necessite une réelle mise en oeuvre. En effet, dans de nombreux logiciels, jouer en parallèle deux fois le même flux musical s'accompagne, en général, d'une augmentation du volume. A défaut d'une telle implementation, cette équivalence devra donc être comprise *modulo* la balance des volumes des pistes.

4.2. Equivalence de flux tuilés

Cette *équivalence observationnelle* se généralise aux flux tuilés en disant que deux flux tuilés t_1 et t_2 sont observationnellement équivalents lorsqu'ils produisent le même effet à l'exécution quel que soit le contexte d'exécution dans lequel on les joue. Cette mise en contexte nous permet de rendre compte des paramètres de synchronisation.

Formellement, en notant $playT$ la fonction qui permet de *jouer* le flux media d'une tuile *entre* les points de synchronisation *pre* et *post*, deux flux tuilés t_1 et t_2 seront équivalents, ce qui sera noté $t_1 \equiv t_2$ lorsque

$$\begin{array}{l}
Tile\ pr_1\ po_1\ m_1 \equiv Tile\ pr_2\ po_2\ m_2 = \\
\quad pr_1 - po_1 == pr_2 - po_2 \wedge \\
\quad \textbf{let}\ d\ = dur\ m_1 - dur\ m_2 \\
\qquad p\ = pr_1 - pr_2 \\
\qquad n_1 = \textbf{if}\ d < 0 \\
\qquad\quad \textbf{then}\ mDelay\ d\ m_1\ \textbf{else}\ m_1 \\
\qquad n_2 = \textbf{if}\ d > 0 \\
\qquad\quad \textbf{then}\ mDelay\ (-d)\ m_2\ \textbf{else}\ m_2 \\
\quad \textbf{in}\ \ \textbf{if}\ p > 0\ \textbf{then}\ n_1\ \text{`}equiv\text{`}\ mDelay\ p\ n_2 \\
\qquad \textbf{else}\ mDelay\ (-p)\ n_1\ \text{`}equiv\text{`}\ n_2
\end{array}$$

On vérifie que deux flux tuilés t_1 et t_2 sont équivalents si et seulement si, pour toute tuiles de silence r_1 et r_2 on a bien

$$tToM\ (r_1 \,\%\, t_1 \,\%\, r_2)\ \texttt{'equiv'}\ tToM\ (r_1 \,\%\, t_2 \,\%\, r_2)$$

4.3. Structure algébrique induite

On vérifie aussi que pour tous flux tuilés t, t_1, t_2 et t_3, les équivalences suivantes sont satisfaites.

$$\begin{array}{rcl}
t_1 \,\%\, (t_2 \,\%\, t_3) & \equiv & (t_1 \,\%\, t_2) \,\%\, t_3 \\
t \,\%\, r\ 0 & \equiv & t \\
r\ 0 \,\%\, t & \equiv & t
\end{array}$$

La première équivalence indique que l'ordre dans lequel on évalue les composants d'un produit tuilé n'a pas d'importance. C'est l'équivalence d'associativité. Les deux suivantes indiquent que la tuile silence $(r\ 0)$ de durée de synchronisation nulle agit comme un élement neutre. Autrement dit, l'équivalence observationnelle induit une structure de *monoïde* sur les tuiles.

Plus encore, on constate aussi que pour tout flux tuilé t on a :

$$\begin{array}{rcl}
t & \equiv & inv(invt) \\
t & \equiv & (t \,\%\, (inv\ t) \,\%\, t) \\
(re\ t) & \equiv & (t \,\%\, (inv\ t)) \\
(co\ t) & \equiv & ((inv\ t) \,\%\, t)
\end{array}$$

La première équivalence indique que l'opération d'inversion est une involution. La seconde, que l'inverse d'un flux est bien un inverse au sens de la théorie des semigroupes [23]. Les quatrième et cinquième équivalences montrent le reset et le co-reset sont en fait des opérations dérivées du produit tuilé et de l'inversion.

On peut poursuivre plus en détail l'étude de cette équivalence est découvrir, comme en théorie des semigroupes inversifs, que les flux tuilés de durée de synchronisation nulle sont les seuls qui satisfont n'importe laquelle des équivalences suivantes :

$$\begin{array}{rcl}
t & \equiv & t \,\%\, t \\
t & \equiv & inv\ t
\end{array}$$

et que, de plus, ils commutent, c'est-à-dire que
$| \ t_1 \,\%\, t_2 \equiv t_2 \,\%\, t_1 \ |$
lorsque t_1 et t_2 sont deux tuiles de durée de synchronisation nulle. La théorie des monoides inversifs [23] nous assurent alors que, modulo équivalence observationnelle, tout flux tuilé t admet un unique inverse $(inv\ t)$. La structure induite est un *monoïde inversif.*

4.4. Codage inverse

Dans ce langage de manipulation de tuiles, nous retrouvons aussi les produits séquentiels et parallèles de Euterpea.

En effet, dans le cas flux musicaux finis, on définit la fonction $mToT$ qui transforme tout flux musical fini m en un flux tuilé par :

$$\begin{aligned} &mToT :: Music\ a \to Tile\ a \\ &mToT\ m = \textbf{let}\ d = dur\ (m) \\ &\qquad\qquad \textbf{in}\ Tile\ 0\ d\ m \end{aligned}$$

On constate que ce codage est injectif vis à vis de l'équivalence observationnelle. On constate de surcroit que ce codage est fonctoriel vis à vis de la composition séquentielle. En effet, pour tout flux musical fini m_1 et m_2 on a bien :
$mToT\ (m_1 :+: m_2) \equiv (mToT\ m_1)\ \%\ (mToT\ m_2)$
Autrement dit, le produit de synchronisation code, via la fonction $mToT$, le produit séquentiel.

Dans le cas de flux (potentiellement) infinis, l'encodage décrit ci-dessous échoue. En effet, le produit séquentiel de deux flux infinis ne peut être définit de façon satisfaisante puisque le second flux se voit en quelque sorte repoussé à l'infini. Il ne pourra être joué.

On retrouve ici une problématique abordée et résolue par la théorie des ω-semigroupes [27] en distinguant les structures finies (les *strings*) et les structures infinies (les *streams*). Pourtant, il apparait que la manipulation conjointe de ces deux types d'objets peut être faites dès lors qu'ils sont tuilés (voir[7] pour plus de détails).

En pratique, on définit la fonction $sToT$ qui transforme tout flux musical (potentiellement) infini m en un flux tuilé par :

$$\begin{aligned} &sToT :: Music\ a \to Tile\ a \\ &sToT\ m = Tile\ 0\ 0\ m \end{aligned}$$

On vérifie que ce codage est injectif sur les flux musicaux infinis. De plus, il est fonctoriel vis à vis de la composition parallèle. En effet, pour tout flux musical (potentiellement) infini m_1 et m_2 on a bien :
$sToT\ (m_1 :=: m_2) \equiv (sToT\ m_1)\ \%\ (sToT\ m_2)$
Autrement dit, le produit tuilé encode, via la fonction $sToT$, le produit parallèle.

Bien entendu, ces deux exemples peuvent sembler artificiels puisque le produit tuilé, lui-même, est définit à partir des produits séquentiels et parallèles. Mais cette objection ne tient pas puisque qu'elle s'appuie sur une implémentation particulière. Tout autre implémentation du produit tuilé satisfera les propriétés énoncées ci-dessus.

5. RESYNCHRONISATION

Un premier jeu de fonctions définit sur les flux tuilés permet de manipuler la position des points de synchronisation positionnés sur un flux musical tuilé.

5.1. Fonctions de resynchronisation

La fonction *resync* permet de déplacer le point de sortie *post* de la synchronisation. De façon duale, la fonction *coresync* permet de déplacer le point d'entrée *pre* de la synchronisation.

$$\begin{aligned} &resync :: Dur \to Tile\ a \to Tile\ a \\ &resync\ s\ (Tile\ pre\ post\ m) = \\ &\quad \textbf{let}\ npost = post + s \\ &\quad \textbf{in if}\ npost < 0 \\ &\qquad \textbf{then}\ Tile\ (pre - npost)\ 0\ (mDelay\ (-npost)\ m) \\ &\qquad \textbf{else}\ Tile\ \ pre\ npost\ m \\ &coresync :: Dur \to Tile\ a \to Tile\ a \\ &coresync\ s\ (Tile\ pre\ post\ m) = \\ &\quad \textbf{let}\ npre = pre + s \\ &\quad \textbf{in if}\ npre < 0 \\ &\qquad \textbf{then}\ Tile\ 0\ (post - npre)\ (mDelay\ (-npre)\ m) \\ &\qquad \textbf{else}\ Tile\ npre\ post\ m \end{aligned}$$

Le comportement de ces fonctions est illustré Figure 7 pour un offset $s > 0$.

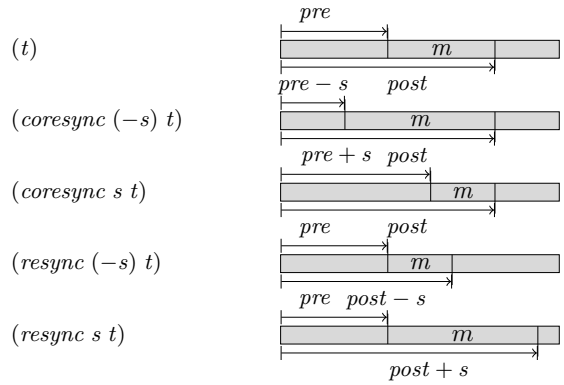

Figure 7. Resynchronisation et co-resynchronisation

5.2. Propriétés de la resynchronisation

Ces fonctions généralisent les fonctions reset *re* et co-reset *co* au sens où on a :

$$\begin{aligned} re\ (Tile\ pr\ po\ m) &= resync\ (pr - po) \\ &\quad\ (Tile\ pr\ po\ m) \\ co\ (Tile\ pr\ po\ m) &= coresync\ (po - pr) \\ &\quad\ (Tile\ pr\ po\ m) \end{aligned}$$

On remarque par ailleurs que ces fonctions induisent des actions du groupe des nombres rationnels muni de l'addition sur l'ensemble des flux tuilés. En pratique, pour tout flux musical tuilé t et tout offset rationnel

a et b on a en effet :

$$\begin{aligned} resync\ 0\ t &= t \\ resync\ a\ (resync\ b\ t) &= resync\ (a+b)\ t \\ coresync\ 0\ t &= t \\ coresync\ a\ (coresync\ b\ t) &= coresync\ (a+b)\ t \end{aligned}$$

En particulier, pour toute tuile t et toute durée s, on constate qu'on a :

$$\begin{aligned} resync\ s\ t &\equiv t \,\%\, r\ s \\ coresync\ s\ t &\equiv r\ s \,\%\, t \end{aligned}$$

Elles sont aussi duales l'une de l'autre au sens des équivalences suivantes :

$$\begin{aligned} resync\ a\ (inv\ t) &\equiv (inv\ (coresync\ a\ t)) \\ coresync\ a\ (inv\ t) &\equiv (inv\ (resync\ a\ t)) \end{aligned}$$

Enfin, leur comportement vis à vis du produit tuilé est décrit par les équivalences observationnelles suivantes. Pour tout flux musical tuilé t_1 et t_2 et tout offset de durée a, on a aussi :

$$\begin{aligned} resync\ a\ (t_1 \,\%\, t_2) &\equiv (resync\ a\ t_1) \,\%\, t_2 \\ coresync\ a\ (t_1 \,\%\, t_2) &\equiv t_1 \,\%\, (coresync\ t_2) \end{aligned}$$

5.3. Fonctions dérivées

Des exemples de fonctions dérivées des fonctions de resynchronisation sont les fonctions insertions de tuiles. Elles sont de deux sortes.

La premiè, fait un *fork* parallèle d'une tuile t_2 dans une tuile t_1 à la position d depuis l'entrée de synchronisation *pre*.

$$\begin{aligned} &insertT :: Dur \to Tile\ a \to Tile\ a \to Tile\ a \\ &insertT\ d\ t_1\ t_2 = coresync\ (-d)\ (re\ t_2 \,\%\, coresync\ d\ t_1) \end{aligned}$$

De façon duale, la seconde, $co-insertion$ fait un *join*.

$$\begin{aligned} &coinsertT :: Dur \to Tile\ a \to Tile\ a \to Tile\ a \\ &coinsertT\ d\ t_1\ t_2 = resync\ (-d)\ (resync\ d\ t_1 \,\%\, co\ t_2) \end{aligned}$$

Le comportement de ces fonctions est décrit Figure 8.

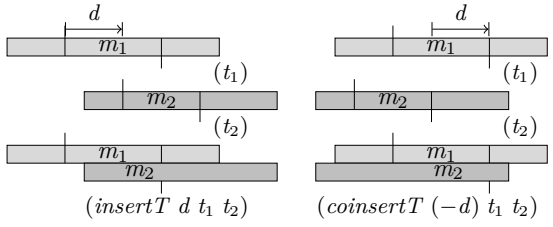

Figure 8. Insertions par Fork et Join.

Remarque. Bien qu'on ait proposé un codage directe de cette fonction, remarquons qu'elle peut aussi se coder simplement à l'aide du produit tuilé, des reset et co-reset, et des tuiles silencieuses. En effet, pour tout flux tuilés t_1 et t_2, et pour tout offset d, on a :

$$\begin{aligned} insertT\ d\ t_1\ t_2 &\equiv r\ d \,\%\, re\ t_2 \,\%\, r\ (-d) \,\%\, t_1 \\ coinsertT\ d\ t_1\ t_2 &\equiv t_1 \,\%\, r\ d \,\%\, co\ t_2 \,\%\, r\ (-d) \end{aligned}$$

5.4. Exemple : croisements temporels

Pour illustrer la puissance de la métaphore du tuilage au sein d'un langage de programmation complet tel que Haskell, on propose ci-dessous le codage d'une fonction *crossing* qui simule, par répétitions et décalages successifs, le *croisement* de deux flux musicaux tuilés.

```
crossing :: Dur → Tile a → Tile a → Tile a
crossing o t1 t2 =
  if ((centerT (t1) > 0)
     ∧ (centerT (t2) > 0)) then
    let v1 = (resync o t1)
        v2 = (resync (−o) t2)
    in (t1 % t2 % (crossing o v1 v2))
  else (t1 % t2)
```

Cette fonction est alors mis en oeuvre dans l'exemple musicale suivant.

```
train1 = liftT (instrument RhodesPiano)
         (r en % t a 3 en % t c 4 en % t e 4 en
              % t d 4 en % r (3 ∗ en))
train2 = liftT (instrument RhodesPiano)
         (t e 4 en % t g 3 en % t a 3 en
                % t a 3 en % r (4 ∗ en))
crossL = crossing (−en) train1 train2
```

L'audition de l'ensemble se fait en executant la commande *playT crossL*. Une ligne de percussion régulière *percL* telle que décrite ci-dessous lancée en parallèle par la commande $playT\ ((re\ percL) \,\%\, crossL)$, permettra d'entendre les phénomènes de décalage de la ligne *crossL*.

6. CONTRACTIONS ET EXPANSIONS

Lors d'une resynchronisation, l'invariant et le flux musicale sous-jacent au flux tuilés. Un jeu de fonctions sensiblement différent est obtenu en préservant la taille de la synchronisation tout en contractant ou étirant le flux musicale sous-jacent. Ce jeu de fonctions est présenté ici.

6.1. Les fonctions de contraction/expansion

La fonction *strech* permet d'étirer et de contracter le flux musical en maintenant la position du point d'entrée de synchronisation *pre* sur le flux musical. de façon duale, la fonction *costretch* étire ou contracte

le flux musical en maintenant le point de sortie de synchronisation *post* sur le flux musical.

La fonction *stretch* est définit par :

$$
\begin{aligned}
&stretch :: Dur \to Tile\ a \to Tile\ a\\
&stretch\ r\ (Tile\ pre\ post\ m) =\\
&\quad assert\ (r > 0)\ (Tile\ \ (pre * r)\ (pre * (r-1) + post)\\
&\qquad\qquad (tempo\ (1/r)\ m))
\end{aligned}
$$

La fonction *costretch* est, quant à elle, définit par :

$$
\begin{aligned}
&costretch :: Dur \to Tile\ a \to Tile\ a\\
&costretch\ r\ (Tile\ pre\ post\ m) =\\
&\quad assert\ (r > 0)\ (Tile\ (post * (r-1) + pre)\ (post * r)\\
&\qquad\qquad (tempo\ (1/r)\ m))
\end{aligned}
$$

Le comportement de ces fonctions est décrit Figure 9 avec un facteur $r > 1$ et en notant (abusivement) $m{*}r$ le flux *tempo* $(1/r)$ m, et m/r le flux *tempo* r m.

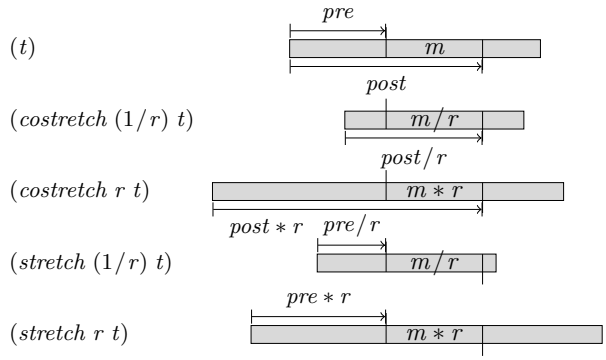

Figure 9. Stretch et co-stretch

6.2. Propriétés des contractions/expansions

De façon analogue aux fonctions de resynchronisation, ces fonctions induisent une action du groupe des nombres rationnels strictement positifs muni de la multiplication sur l'ensemble des flux tuilés. Pour tout flux musical tuilé t et tout offsets a et b, on a :

$$
\begin{aligned}
stretch\ 1\ t &= t\\
stretch\ a\ (stretch\ b\ t) &= stretch\ (a * b)\ t\\
costretch\ 1\ t &= t\\
costretch\ a\ (costretch\ b\ t) &= costretch\ (a * b)\ t
\end{aligned}
$$

Ces deux fonctions sont duales l'une de l'autre au sens des équivalences suivantes :

$$
\begin{aligned}
stretch\ a\ (inv\ t) &\equiv (inv\ (costretch\ a\ t))\\
costretch\ a\ (inv\ t) &\equiv (inv\ (stretch\ a\ t))
\end{aligned}
$$

Enfin, le comportement de ces fonctions vis à vis du produit synchronisé est décrit par les equations suivantes. Pour tout flux musical tuilé t_1 et t_2 et tout facteur $a > 0$, on a :

$$
\begin{aligned}
&stretch\ a\ (t_1\ \%\ t_2) \equiv\\
&\quad (tempoT\ (1/a)\ t_1)\ \%\ (stretch\ a\ t_2)
\end{aligned}
$$

$$
\begin{aligned}
&costretch\ a\ (t_1\ \%\ t_2) \equiv\\
&\quad (costretch\ a\ t_1)\ \%\ (tempoT\ (1/a)\ t_2)
\end{aligned}
$$

6.3. Exemples : génération rythmique

L'utilisation de *stretch* et *costretch* peut être illustré par les transformations rythmiques déjà évoquées dans [15].

$$
\begin{aligned}
&march = t\ c\ 4\ qn\ \%\ r\ qn\ \%\ t\ g\ 4\ qn\ \%\ r\ qn\\
&waltz\ \ = costretch\ (2/3)\ march\\
&tumb = costretch\ (5/4)\ march
\end{aligned}
$$

Le rythme initial *march*, dans une métrique à deux temps, consiste à jouer un *do* sur le premier temps et un *sol* sur le second temps.

Dans le rythme *waltz*, dans une métrique à trois temps, les notes se retrouvent jouées sur le deuxième et le troisième temps de la durée de synchronisation. C'est donc effectivement une ligne de basse pour une valse.

Dans le rythme *tumb*, dans une métrique à quatre temps, le *do* est joué sur le 4ème temps de la mesure qui précède, et le *sol* sur la levée du 2ème temps. C'est là une ligne de basse typique de la salsa cubaine : le *tumbao*. Ces trois exemples sont décrits Figure 10.

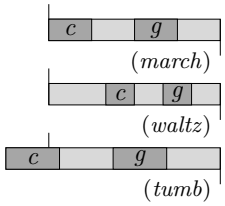

Figure 10. Cellules rythmiques engendrées par contraction/expansion

Joués en parallèle à une structure rythmique répétitive marquant le début de chaque mesure, c'est à dire le point *pre*, on peut écouter ces cellules à l'aide du code suivant :

$$
\begin{aligned}
bass &= liftT\ (instrument\ Percussion)\\
&\quad (Tile\ 0\ wn\ (perc\ AcousticBassDrum\ wn))\\
hiHat &= liftT\ (instrument\ Percussion)\\
&\quad (Tile\ 0\ (1/8)\ (perc\ ClosedHiHat\ (1/8)))\\
bassL &= bass\ \%\backslash\ re\ bassL\\
hiHatL &= repeatT\ 4\ hiHat\ \%\backslash\ re\ hiHatL\\
percL &= re\ bassL\ \%\ hiHatL\\
testW &= playT\ (re\ bassL\ \%\ tempoT\ (3/4)\\
&\quad\qquad (re\ hiHatL)\ \%\ repeatT\ 4\ waltz)\\
testS &= playt\ (re\ bassL\ \%\ re\ hiHatL\\
&\quad\qquad \%\ repeatT\ 4\ tumb)
\end{aligned}
$$

Remarque. Ces exemples sont construits à l'aide d'un opérateur de produit $\%\backslash$ qui, en limitant les superpositions vers le passé, permet de définir facilement

des tuiles récursives. On trouvera ci-dessous, une première discussion sur les définitions récursives tuilés. Une discussion plus poussé sur cette problèmatique délicate pourra être trouvé dans [12].

7. TUILES RÉCURSIVES

Dans les exemples ci-dessus, nous avons vu comment les structures de contrôle classique des langages de programmation permettent de définir *algorithmiquement* des flux tuilés finis complexes.

Le langage Haskell, reposant sur un principe d'évaluation paresseuse, permet aussi de définir des objets potentiellement infinis qui sont évalués à la demande. Nous souhaiterions pouvoir utiliser cette caractéristique pour définir des tuiles potentiellement infinies.

Par exemple, étant donné une tuile finie t, on souhaiterait pouvoir définir une tuile x de support musical infini, via une equation de la forme

$$x = t \,\%\, (re\ x)$$

Dans une telle equation, l'appel récursif se fait sur le *reset* de la variable x afin de produire une tuile dont la distance de synchronisation serait celle de la tuile t. Pourtant, malgré cette précaution, l'évaluation de cette équation en Haskell boucle !

En effet, le modèle des flux tuilés est ainsi fait que les superpositions en amont du point de synchronisation *pre* peuvent provenir de tuiles situées arbitrairement en aval dans une suite de produit synchronisé. Notre implémentation boucle donc sur une telle équation car elle doit dérouler toute les répétitions de x pour en connaitre les anticipations : le typage de la tuile x ainsi définit par équation échoue à être calculé par Haskell.

Dans [21], des conditions nécessaires et suffisantes, calculables, sont décrites pour résoudre cette récursion. Néanmoins, elles sont proposées dans une définition de T-calcul pure, particulièrement limitée. Il ne contient pas de structure de contrôle telle que les conditionnelles. Dans l'implémentation du T-calcul proposée ici, qui hérite de toute l'expressivité d'Haskell, le calcul du type d'un flux tuilé définit par équation est, en toute généralité, indécidable.

Pour remédier à cela, nous proposons un nouveau produit synchrone, partiel, qui résout ce calcul de type potentiellement cyclique en « coupant » les anticipations qui y conduisent. Plus précisément, on définit le produit restreint $\%\backslash$ suivant :

$$(\%\backslash) :: Tile\ a \to Tile\ a \to Tile\ a$$
$$Tile\ pr_1\ po_1\ m_1\ \ \%\backslash\ \sim(Tile\ pr_2\ po_2\ m_2) =$$
$$\quad Tile\ pr_1\ po_1\ (m_1 :=: mDelay\ po_1\ (dropM\ pr_2\ m_2))$$

Remarque. Le $\sim$ dans cette définition est une technicité qui permet de gouverner l'évaluation paresseuse.

On vérifie facilement qu'une équation de la forme

$$x = t \,\%\backslash\, (re\ x)$$

peut maintenant être typée par Haskell. Le type de la tuile x est bien le type de la tuile t. En effet, l'anticipation induite par $(re\ x)$ a été supprimée dans le produit restreint et sa durée de synchronisation est nulle grace au reset. Les marqueurs d'entrée *pre* et de sortie *post* de toute solution sont donc uniquement déterminés par ceux de la tuile t. Une généralisation de ce codage, plus souple, est proposée dans [12].

8. CONCLUSION

Le T-calcul en Euterpea apparait clairement, via les nombreuses propriétés algébriques qu'il satisfait, comme un formalisme particulièrement robuste pour une description hiérarchique de la structure temporelle des flux media temporisés. Son expérimentation pour la modélisation musicale est en cours.

Bien entendu, l'implémentation proposée ici ne traite que des flux musicaux symboliques. Une intégration des flux audio tuilés reste à mettre en oeuvre. Elle pourrait s'appuyer sur la *libTuiles* déjà réalisée [2, 22].

Une telle extension offrirait sans doute un gain de productivité important pour la création et la production de musique électroacoustique. Une bonne partie du mixage, parfois fastidieux et répétitif, peut en effet être factorisée grace à la métaphore du tuilage : chaque tuile audio intègre une fois pour toute, dans ses points de synchronisation, toute l'information necessaire pour positionner, *avant* ou bien *après*, les autres tuiles audio.

Remarquons aussi que le langage proposé ici n'est destiné qu'à une manipulation « hors temps » des structures musicales. Bien sur, en toute généralité, on ne peut décider de la terminaison de ces programmes. Remarquons cependant que, dès lors qu'un flux tuilés est typé, il est de durée de synchronisation fini. La fonction $playT$ de lecture de ce flux terminera donc puisqu'elle ne parcourt les flux qu'entre les marqueurs de synchronisation *Pre* et *Post*.

On peut mentioner enfin qu'une théorie des langages adaptée à la description et à la manipulation de sous-ensembles de monoïdes inversifs est actuellement en cours de développement. Elle offre des outils traditionnels de manipulation des langages, qu'ils soient logiques [17], algébriques [14, 16] ou qu'ils s'appuient sur la théorie des automates [19, 18, 16]. Autrement dit, en s'appuyant sur cette théorie, il sera possible de développer les outils de spécification, d'analyse et de validations qui pourront accompagner efficacement les outils de modélisations par tuilage.

9. REFERENCES

[1] C. Agon, J. Bresson, and G.Assayag. *The OM composer's Book, Vol.1 & Vol.2.* Collection Musique/Sciences. Ircam/Delatour, 2006.

[2] F. Berthaut, D. Janin, and M. DeSainteCatherine. libTuile : un moteur d'exécution multi-échelle de pro-

cessus musicaux hiérarchisés. In *Actes des Journées d'informatique Musicale (JIM)*, 2013.

[3] F. Berthaut, D. Janin, and B. Martin. Advanced synchronization of audio or symbolic musical patterns : an algebraic approach. *International Journal of Semantic Computing*, 6(4) :409–427, 2012.

[4] J. Bresson, C. Agon, and G. Assayag. Visual Lisp / CLOS programming in OpenMusic. *Higher-Order and Symbolic Computation*, 22(1), 2009.

[5] P. Desain and H. Honing. LOCO : a composition microworld in Logo. *Computer Music Journal*, 12(3) :30–42, 1988.

[6] A. Dicky and D. Janin. Embedding finite and infinite words into overlapping tiles. Research report RR-1475-13, LaBRI, Université de Bordeaux, 2013.

[7] P. Hudak. An algebraic theory of polymorphic temporal media. In *Proceedings of PADL'04 : 6th International Workshop on Practical Aspects of Declarative Languages*, pages 1–15. Springer Verlag LNCS 3057, June 2004.

[8] P. Hudak. A sound and complete axiomatization of polymorphic temporal media. Technical Report RR-1259, Department of Computer Science, Yale University, 2008.

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

[10] P. Hudak and D. Janin. Tiled polymorphic temporal media. Research report RR-1478-14, LaBRI, Université de Bordeaux, 2014.

[11] P. Hudak, T. Makucevich, S. Gadde, and B. Whong. Haskore music notation – an algebra of music. *Journal of Functional Programming*, 6(3) :465–483, May 1996.

[12] D. Janin. Quasi-recognizable vs MSO definable languages of one-dimensional overlapping tiles. In *Mathematical Found. of Comp. Science (MFCS)*, volume 7464 of *LNCS*, pages 516–528, 2012.

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

[14] D. Janin. Algebras, automata and logic for languages of labeled birooted trees. In *Int. Col. on Aut., Lang. and Programming (ICALP)*, volume 7966 of *LNCS*, pages 318–329. Springer, 2013.

[15] D. Janin. On languages of one-dimensional overlapping tiles. In *Int. Conf. on Current Trends in Theo. and Prac. of Comp. Science (SOFSEM)*, volume 7741 of *LNCS*, pages 244–256. Springer, 2013.

[16] D. Janin. Overlaping tile automata. In *8th International Computer Science Symposium in Russia (CSR)*, volume 7913 of *LNCS*, pages 431–443. Springer, 2013.

[17] D. Janin. Walking automata in the free inverse monoid. Research report RR-1464-12, LaBRI, Université de Bordeaux, 2013.

[18] D. Janin. Towards a higher dimensional string theory for the modeling of computerized systems. In *Int. Conf. on Current Trends in Theo. and Prac. of Comp. Science (SOFSEM)*, volume 8327 of *LNCS*, pages 7–20. Springer, 2014.

[19] 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 *ACM Workshop on Functional Art, Music, Modeling and Design (FARM)*, pages 23–34. ACM Press, 2013.

[20] D. Janin, F. Berthaut, and M. DeSainteCatherine. Multi-scale design of interactive music systems : the libTuiles experiment. In *Sound and Music Computing (SMC)*, 2013.

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

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

[23] S. Letz, Y. Orlarey, and D. Fober. Real-time composition in Elody. In *Proceedings of the International Computer Music Conference*, pages 336–339. ICMA, 2000.

[24] D. Perrin and J.-E. Pin. *Infinite Words : Automata, Semigroups, Logic and Games*, volume 141 of *Pure and Applied Mathematics*. Elsevier, 2004.