LaBRI, CNRS UMR 5800
Laboratoire Bordelais de Recherche en Informatique

Rapport de recherche RR-1466-13

The T-Calculus : towards a structured programing of (musical) time and space

July 29, 2013

David Janin[1], Florent Berthaut[1], Myriam DeSainte-Catherine[1], Yann Orlarey[2], Sylvain Salvati[1,3]

[1] Université de Bordeaux
LaBRI UMR 5800
351, cours de la libération
F-33405 Talence, FRANCE

[2] GRAME
Centre Nat. de Création Musicale
11, Cours de Verdun
F-69002 Lyon, FRANCE

[3] Inria Bordeaux - Sud-Ouest
200, avenue de la Vieille Tour
F-33405 Talence, FRANCE

Contents

The T-Calculus : towards a structured programing of (musical) time and space

David Janin[1]*, Florent Berthaut[1], Myriam DeSainte-Catherine[1], Yann Orlarey[2], Sylvain Salvati[1,3]

[1] Université de Bordeaux
LaBRI UMR 5800
351, cours de la libération
F-33405 Talence, FRANCE

[2] GRAME
Centre Nat. de Création Musicale
11, Cours de Verdun
F-69002 Lyon, FRANCE

[3] Inria Bordeaux - Sud-Ouest
200, avenue de la Vieille Tour
F-33405 Talence, FRANCE
Corresp. author: janin@labri.fr

July 29, 2013

Abstract

In the field of music system programming, the T-calculus is a proposal for combining space modeling and time programming into a single programming feature: spatiotemporal tiled programming. Based on a solid algebraic model, it aims at decomposing every operation on musical objects into the sequence of a synchronization operation that describes how objects are positioned one with respect the other, and a fusion operation that describes how their values are then combined. A first simple version of such a *tiled calculus* is presented and studied in this paper.

1 Introduction

The complex structure space of music

The structure of music is complex. Being polyphonic, it is structured in some musical space (P). Being rhythmic, it is structured in some musical time (T). It also combines various levels of abstraction (A), from low level audio signal

*partially funded by the project INEDIT, ANR-12-CORD-009

processing to high level concert movements combination, upon which may depend the space and time scales. It can even be interactive (I), that is, subject to changes depending on the input of several independent (or loosely coupled) musical sources.

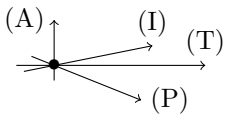

Figure 1: The 4D structure space of computational music

This observation leads to the development of application specific languages, dedicated to music programming, that are based on generic and well-defined programming language paradigms. For instance, the versatile DSP programming language *Faust* [6] is based on the synchronous programming paradigm of languages such as Lustre or Esterel. The domain-specific language *Euterpea* [8], embedded in the typed functional language *Haskell* [9], is another example.

These languages provide strong abstraction design principles that can be used efficiently when programming musical applications. Using such mathematically well-defined programming languages increases both the efficiency of the design and implementation process and the reliability of the resulting application or system.

Modeling space and programming time

We aim at contributing to the development of these languages. We are more specifically concerned with the following problem: coping with the combined modeling/programming of musical time and space.

It is a common observation that music writing involves both sequential composition (in time) and parallel composition (in space) of musical objects. Programming these two features will often amount to:

- ▷ *space modeling (data structure)*: creating a (finite) vector of musical objects to be played in parallel, with one local player per musical objects,
- ▷ *time programming (control flow)*: creating a (potentially infinite and evolving) list of musical objects to be played in sequence, with a single global player for that list.

A priori, from a strict logical point of view, this is however *not* a necessity.

For instance, in Hudak's proposal of polymorphic temporal media [7], the spatial and temporal dimensions are already quite mixed. This tells us that there is no real necessity no need to distinguish between spatial dimensions that describe parallelism and time dimensions that describe execution flows. These are just dimensions that together create the spatiotemporal space within which music evolves.

Some recent modeling experiments [10, 1] even show that there *is* a mathematically well-founded model: tiling semigroups [15, 16], that allows for the description of combinations of musical objects in both dimensions with a *single* operator: the *tiled product*.

Towards (musical) model based programming

Our purpose is to examine to which extent and for what benefit such a tiled product can be integrated in a programing language as a built-in programming feature.

More precisely, every musical object is seen as a *partial* function $m : \mathcal{S} \to \mathcal{D}$ from some spatiotemporal space $\mathcal{S}$ into some set $\mathcal{D}$ of musical values. The domain $dom(m) \subseteq \mathcal{S}$ of such a musical object m describes the *spatiotemporal structure* of that object and, for every $x \in dom(m)$, the value $m(x) \in \mathcal{D}$ describes the local *musical state* of that object at position x.

Then, given two musical objects $m_1 : \mathcal{S} \to \mathcal{D}_1$ and $m_2 : \mathcal{S} \to \mathcal{D}_2$, every operation on these musical objects that aims at defining a new musical object $F(m_1, m_2) : \mathcal{S} \to \mathcal{D}_3$ is decomposed as a sequence of two primitive operations:

- ▷ *synchronization:* defining the domain $dom(F(m_1, m_2))$ of the resulting musical object as some generic combination $dom(m_1) \oplus dom(m_2)$ that tells how the domains $dom(m_1)$ and $dom(m_2)$ are translated to be positioned one relative to the other with possible overlaps,
- ▷ *fusion:* defining how the new musical states are defined on each position of the resulting domain from the local musical states of the translated musical objects m_1 and m_2.

Doing so, we somehow aim at generalizing to spatiotemporal structures the notion of spatial programming that is already emerging in bioinformatics system modeling or musical structure analysis [2, 3].

A study focussed on discrete and regular musical objects

In this paper, the musical objects under study are discrete, e.g. notes or chords, and regular, i.e. with integer durations. Possible extensions of the work presented here, either to musical objects with continuous domains as in audio processing, or to irregular musical objects built on atomic elements with arbitrary duration, as in [7], are left for further studies.

2 Strings and streams for music modeling

In this section, we review some of the basic features of the (finite) string and (infinite) stream data types that can be used in music programming. Then, by taking some examples from music, we show how strings and streams fail to satisfy some compositional properties music system designers may expect when modeling music.

From now on, let $\mathcal{D}$, $\mathcal{D}_1$, $\mathcal{D}_2$, etc., be some alphabet types. Elements of a type $\mathcal{D}$ will be the elements of the strings and streams that we handle.

As far as interpretation in music is concerned elements of type $\mathcal{D}$ are treated, to keep things simple, as musical atoms of duration one. It follows that the index of every element (musical atom) in a string, a stream, or even a tiled stream as defined in the next section, is also seen as the (relative) *date* at which that musical atom *starts*. The first element in a string or a stream, at position 0, thus starts at the (relative) date 0.

2.1 Strings in music programming

Strings (or equivalent data types) are probably one of the basic data-structure one may use in music programming as shown for instance in the *libAudioStream* [19]. Since we propose below to extend this data type, let us first review the definitions related to strings.

By $\mathcal{D}$-strings we mean here any finite (possibly empty) list of elements of type $\mathcal{D}$. A $\mathcal{D}$-string is thus a mapping of the form $m : [0, k-1] \to \mathcal{D}$ where $[x, y]$ refers to as the interval set of integers $\{x, x+1, \cdots, y\}$ when $x \leq y$ and the empty set otherwise, and where $k \in \mathbb{N}$, from now on denoted by $|m|$ is the length of the string.

The concatenation product $(m_1 \cdot m_2)$ of two $\mathcal{D}$-strings m_1 and m_2 (denoted by :+: in [7]) is defined as the string obtained by putting string m_2 *right after* string m_2 hence with $|m_1 \cdot m_2| = |m_1| + |m_2|$, that is, for every $k \in [0, |m_1| + |m_2| - 1]$,

$$(m_1 \cdot m_2)(k) = \begin{cases} m_1(k) & \text{if } 0 \leq k < |m_1| \\ m_2(k - |m_1|) & \text{if } |m_1| \leq k < |m_1| + |m_2| \end{cases}$$

Example 2.1 Our running musical example is the following bebop tune, depicted in Figure 2, that has already been considered in [10]. A simple analysis

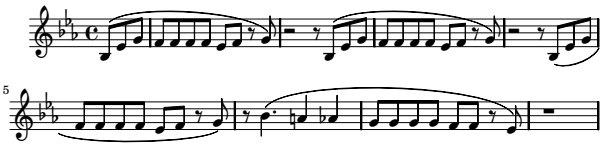

Figure 2: *My little suede shoes* (C. Parker, 1951)

of the musical structure of that song shows that it is composed with three repetitions of a first melodic line, depicted in Figure 3, that is followed by a second melodic line, depicted in Figure 4. Given a data type $\mathcal{D}$ that simply corresponds

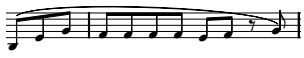

Figure 3: A first melodic line m_1

to notes and silences of duration one eighth, we can encode these melodic lines as two $\mathcal{D}$-strings m_1 and m_2 with the respective lengths $|m_1| = 11$ and $|m_2| = 15$.

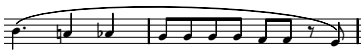

Figure 4: A second melodic line m_2

Of course, for the second melodic line, we assume that there is some special symbol in $\mathcal{D}$ that merely means "keep on playing the same note" in order to encode the quarter notes and the dotted quarter note.

The entire melodic line is then encoded by the string m defined by $m = m_1 \cdot cs_5 \cdot m_1 \cdot cs_5 \cdot m_1 \cdot cs_1 \cdot m_2$ where the constant strings cs_k describe strings of silences of length k. The structure of such a concatenation can be depicted as in Figure 5 with a rhythmical count (in numbers of eighths) starting at zero.

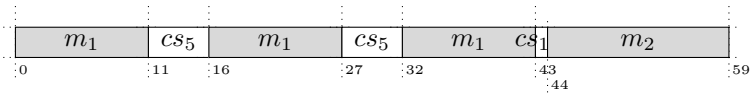

Figure 5: The structure of the complete melodic line

An immediate remark is that, in this concatenation example of the first and second melodic lines, we have to insert silences of various length. Doing so, the general structure of the entire melodic line, stated as "three times melody m_1 followed by melody m_2" is a little lost.

Continuing our example, if we want to play twice that melody, then we have to construct the melodic line mm defined by $mm = m \cdot cs_5 \cdot m$. Again, this encoding is definitely lacking musical sense. No musician will ever pay attention that 5 eighth silences that are needed here. Indeed, musicians synchronize on music bars.

Everything looks as if the *musical concatenation* of melodies implicitly described in Figure 2 *is not* a simple string concatenation.

2.2 On musical anacrusis and related notions

In musical terms, as already observed in [10], both the first and the second melodies are built upon the notion of *anacrusis*, that is, a note or series of notes that comes before the first complete measure of a melody. It prepares the listener's ears for the first beat of the next measure (also called downbeat).

This notion leads to distinguishing the first note of a melody, referred to as its *effective start*, from its first downbeat, referred to as its *logical start*. By logical we mean here logical with respect to the underlying time signature.

Similar phenomena also occur at the end of the melodic lines depicted in Figure 3 and Figure 4. This is especially true in Bebop style. The last notes

of these melodies are played right *before* the bar in order to emphasize the next strong beat, even though silent.

In other words, some amount of silence *does* conclude the underlying rhythm. There is thus a need to distinguish the *effective stop* from the (rhythmically) *logical stop*.

A more accurate description of these melodic lines is depicted in Figure 6 with an entire measure of silence added at their ends.

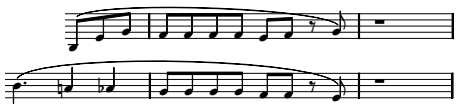

Figure 6: The completed second melodic line

The last four measures of the entire melody (with an added anacrusis) are then better described, with some overlaps, as depicted in Figure 7. In that figure,

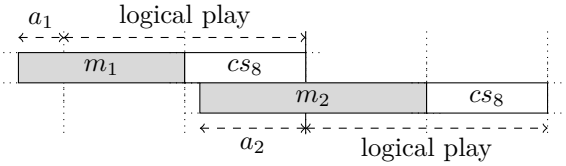

Figure 7: Sequential composition of plays

anacrusis are marked a_1 for m_1 and a_2 for m_2. It shows that the meaning of the timed preposition *before* (resp. *after*) for the string data type *is not* the meaning it has in music.

2.3 Streams in music programming

Streams, that is, (potentially) infinite sequences of values, are often used in music to model audio signals as in *Faust* [6]. They also appear in *Haskell* [9] via the lazy evaluation mechanism. Again, as we propose below to extend this data type, let us first review the definitions related to streams.

By $\mathcal{D}$-streams we mean here an infinite list of elements of some type $\mathcal{D}$. A $\mathcal{D}$-stream is thus a mapping $s : \mathbb{N} \to \mathcal{D}$. The main operation on streams is the parallel product (denoted by :=: in [7]). For every $\mathcal{D}_1$-stream s_1 and $\mathcal{D}_2$-string s_2, the parallel product

$$s_1|s_2 : \mathbb{N} \to \mathcal{D}_1 \times \mathcal{D}_2$$

of the two streams s_1 and s_2 is defined, for every $k \in \mathbb{N}$, by

$$(s_1|s_2)(k) = (s_1(k), s_2(k))$$

Observe that, we have $(s_1|s_2) \simeq (s_2|s_1)$, i.e. the parallel product is, up to isomorphism, commutative.

With both strings and streams, we also have a mixed product (denoted by ::) that takes one $\mathcal{D}$-string m and one $\mathcal{D}$-stream s as inputs and that produces the $\mathcal{D}$-stream $m :: s$ obtained by placing the string m in front of the stream s, i.e. the stream $m :: s$ is defined, for every $k \in \mathbb{N}$, by

$$(m :: s)(k) = \begin{cases} m(k) & \text{if } 0 \leq k < |m| \\ s(k - |m|) & \text{if } |m| \leq k \end{cases}$$

Example 2.2 (2.1 continued) Going back to our musical example, given a constant $\mathcal{D}$-stream $null_{\mathcal{D}}$ that is defined as the silent stream defined, for every $k \in \mathbb{N}$, by $null_{\mathcal{D}}(k) = 0_{\mathcal{D}}$ for some elementary silence $0_{\mathcal{D}}$. Then, the operation defined above allows for converting the entire melody m in Figure 2 into the $\mathcal{D}$-stream $s = m :: null_{\mathcal{D}}$.

A priori, playing the melodic stream s in together with another stream s' – say a bass line – amounts then to defining the parallel stream $p = s|s'$. However, for reasons similar to the string case examples with anacrusis, such a product is not satisfactory because it assumes that the two streams effectively start at the same logical time.

It may be the case, for instance, that the stream s' of the bass line starts right after the first bar. In that case, what we want to build is defined as the parallel stream $p = s|(cs_3 :: s')$, the length of the silence cs_3 being defined by some comparison between the musical nature of s and s'. Such a situation is

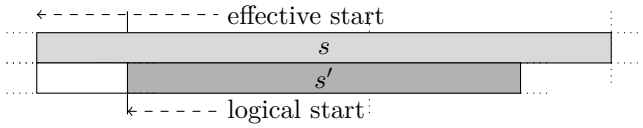

Figure 8: Parallel composition of plays

depicted in Figure 8.

It may also be the case that the bass line actually starts one measure before. Then, the correct parallel play is rather defined as $p = (cs_5 :: s)|s'$.

Again, we fail to capture the accurate musical nature of the melodic lines we aim at encoding. The parallel product of two streams *does not* match the intended notion of parallel product of melodies a composer or a musician may wish to use. Indeed, with such an encoding the intended parallel composition depends on the properties of the composed streams s and s': it is not uniformly defined as one would expect.

In other words, the meaning of the prepositional phrase *at the same time* that is induced by the stream data type *is not* the more subtle meaning it may have in music.

3 From strings and streams to tiled streams

In the examples studied in the previous section, either with strings and streams, everything looks as if we need additional information (e.g. synchronization markers) to tell how these structures are composed either in a sequential or in a parallel musically accurate way.

This is done by embedding both strings and streams types into more general structures: tiled streams, with an associated tiled product. Then, it is shown that both strings catenation or streams parallel product are particular cases of tiled stream product.

3.1 Tiled streams

Simply said, a tiled stream is a *bi-infinite sequence* of elements with *two distinguished synchronization marks* positioned *between* elements.

Formally, a *tiled $\mathcal{D}$-stream* is modeled as a mapping $t : \mathbb{Z} \to \mathcal{D}$ of values of type $\mathcal{D}$ and a synchronization length $d \in \mathbb{N}$. The first synchronization mark in a tiled $\mathcal{D}$-stream is always assumed to be right before the element at index 0. The second synchronization mark is always assumed to be at distance d from the first, that is, right before the element at index d.

In order to keep notation simple, we just write $t : \mathbb{Z} \to \mathcal{D}$ for such a $\mathcal{D}$-stream and we write $d(t) \in \mathbb{N}$ for its synchronization length/duration.

Remark 3.1 Playing in real time a tiled stream t will amount to choosing a real date t_0, and playing, one after the other, the musical event $t(k)$ at the date $t_0 + k$ for all $k \in \mathbb{Z}$. In other words, indices in tiled streams correspond to (possibly negative) date relative to origin index 0. As for strings or streams, when element indices are seen as dates, they are always understood as the start date of the corresponding element in the tiled stream.

Definition 3.2 (Synchronization and realization intervals) *We define, over integers seen as (relative) dates, the* synchronization interval *of a tiled stream t to be the interval $[0, d(t)]$.*

Then, assuming that $\mathcal{D}$ is equipped with a distinguished element $0_{\mathcal{D}}$ that acts as a default (silent) value. We define a realization interval *of the tiled stream t, to be any non empty (integer) interval $[b, e] \subseteq \mathbb{Z}$, i.e. namely beginning and ending dates, such that $[0, d] \subseteq [b, e]$ with $t(k) = 0_{\mathcal{D}}$ for every $k \notin [b, e]$.*

This notion of realization interval is extended to the infinite by completing $\mathbb{Z}$ with a least element $-\infty$ and a greatest element $+\infty$.

Such a tiled stream with finite realization interval, as in most examples pictured later, is depicted in Figure 9.

Remark 3.3 Pursuing the analogy with musical notation, synchronization marks can be seen as bars in music score, all events or notes before the first mark seen as anacrusis, all events or notes after the second being conclusive.

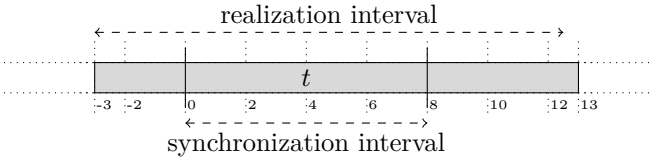

Figure 9: A typical tiled stream with anticipated start and late end

The tiled product, defined below, tells how two tiled streams are to be positioned one *after* the other. Since tiled streams are bi-infinite this implies that the tiled product induces overlaps. The tiled product is thus both a sequential and a parallel product.

Formally, for every tiled $\mathcal{D}_1$-stream t_1 and $\mathcal{D}_2$-string t_2, the tiled product $t_1 \,;\, t_2$ of the two tiled streams t_1 and t_2 is the $\mathcal{D}_1 \times \mathcal{D}_2$-tiled stream defined with the synchronization duration $d(t_1 \,;\, t_2) = d(t_1) + d(t_2)$ and

$$(t_1 \,;\, t_2)(k) = (t_1(k), t_2(k - d(t_1)))$$

for every $k \in \mathbb{N}$. An example of tiled product is depicted in Figure 10. It is an

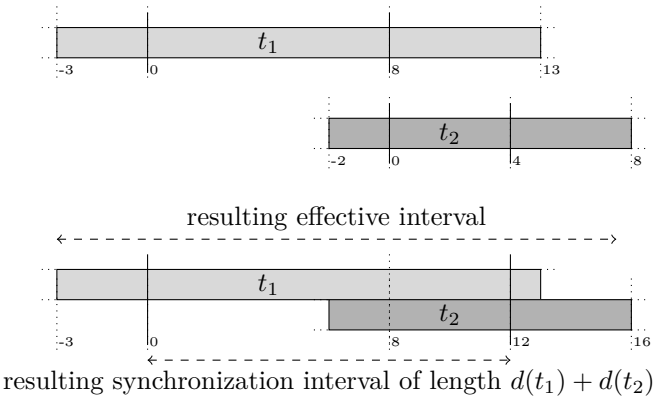

Figure 10: The tiled product of t_1 and t_2

easy exercise to check that, up to the associativity isomorphism on basic types cartesian product, the tiled product is associative.

Remark 3.4 The fact that the two synchronization marks may coincide in a tiled stream only indicates that the role of that musical part is to be synchronized (on its first synchronization mark) before being launched/played. It does not take part in the underlying, more global, synchronization/time positioning process.

3.2 From musical strings to musical tiled streams

Going back to Example 2.1, let m_1 and m_2 be the $\mathcal{D}$-string defined in Section 2.1. We encode the first melodic line in Figure 3 by the tiled stream t_1 defined, for every $k \in \mathbb{Z}$, by

$$t_1(k) = \begin{cases} m_1(k+3) & \text{if } -3 \leq k < |m_1| - 3 \\ 0_{\mathcal{D}} & \text{(otherwise)} \end{cases}$$

with synchronization duration $d(t_1) = 16$, and we encode the second melodic line in Figure 4 by the tiled stream t_2 defined, for every $k \in \mathbb{Z}$, by

$$t_2(k) = \begin{cases} m_2(k+7) & \text{if } -7 \leq k < |m_2| - 7 \\ 0_{\mathcal{D}} & \text{otherwise} \end{cases}$$

with synchronization duration $d(t_2) = 16$. Then, defining the entire melodic line in Figure 2 just amount to perform the tiled products:

$$t = t_1 \, ; t_1 \, ; t_1 \, ; t_2$$

that simply encodes our initial musical analysis, i.e. three times the first melodic line followed by one time the second melodic line.

Remark 3.5 The resulting model, that can be depicted much as in the style of Figure 7 above, has four parallel voices. Strictly speaking, we need now to merge these four voices into a single one, possibly, in the general case, by allowing chords instead of single notes. Such a merge operator is defined in Section 3.5 below.

As detailed in Section 4.3 below (see Example 4.9), this merge is equivalently defined by the T-calculus program $p = t_1 + t_1 + t_1 + t_2$.

3.3 Synchronization reset and co-reset

We also aim at embedding streams into tiled streams. In order to do so, we extend the tiled stream data type with two additional operators on tiled streams that essentially amount to (re)set the synchronization length of tiled streams to zero.

Definition 3.6 (Tiled stream reset and co-reset) *For every tiled $\mathcal{D}$-stream t, the synchronization reset $R(t)$ and the synchronization co-reset $L(t)$ of the tiled $\mathcal{D}$-stream t are defined by $d(R(t)) = d(L(t)) = 0$ and, for every $k \in \mathbb{Z}$, by*

$$R(t)(k) = t(k) \text{ and } L(t)(k) = t(k - d(t))$$

These operators are named L and R in accordance to their link with left and right Green's classes in (inverse) semigroup theory (see e.g. [12] for an example and [17] for the general theory). They are depicted in Figure 11.

The most basic application of reset and co-reset operators is the generic *join* and *fork* operators one can derive from them as depicted in Figure 12. The *fork*

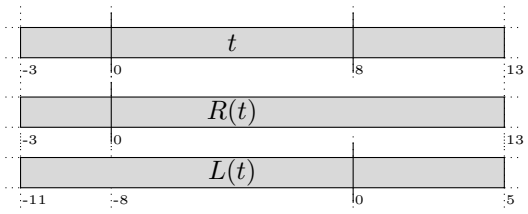

Figure 11: Reset $R(t)$ and co-reset $L(t)$ of a tiled stream t.

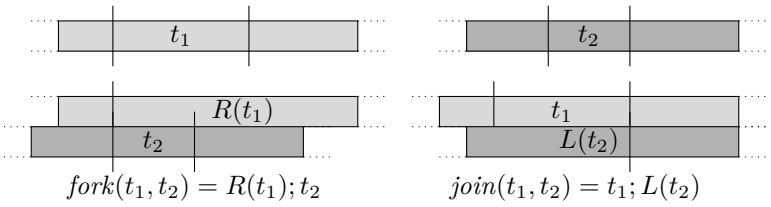

Figure 12: Derived *fork* and *join* operators.

and *join* operators are not commutative. Indeed, the tiled stream $fork(t_1, t_2)$ inherits from the logical interval of t_2 while the tiled stream $fork(t_2, t_2)$ inherits from the logical interval of t_1. A similar remark holds for $join(t_1, t_2)$ and $join(t_2, t_1)$.

Though not explicitly used in the sequel, the type of *asymmetric* parallel synchronization that is handled by the fork or the join operators is implicitly used in most examples below.

3.4 From musical streams to musical tiled streams

We are now ready to solve the compositionality problems encountered in Section 2.3 when modeling music with streams, illustrated in Section 2.3.

Let t be the tiled stream t defined in Section 2.3 that encodes the melodic line in Figure 2. Let t' be another tiled stream t' that models some bass line.

Assuming that $t'(0)$ models the first eighth note or silence of the first bar of that bass line, then, the parallel play p of both the melodic line and the bass line is simply modeled by

$$p = R(t)\,;R(t') = R(fork(t, t')) = R(fork(t', t))$$

In other words, since the logical start of the lead melody and the bass line are encoded in their respective tiled stream encodings, their parallel product just amounts to synchronizing them on their logical starts.

Remark 3.7 More generally, with the proposed model of tiled streams, we eventually internalize in each model some synchronization information that is

needed to position in time two melodic lines, be them put in sequence, as with strings, or in parallel as with streams. Doing so, the tiled product is a composition operator that has features of both a sequential product and a parallel product.

3.5 Strings and streams embeddings

In [7] it is proved that sequential and parallel products together with silences are enough to encode any musical value/musical score. In this section, we show that both strings and streams can be embedded into tiled streams thus inheriting of such a modeling completeness property.

Prior to defining and proving the embeddings, we first need to formally define the *merge* function. For such a purpose, we assume that every basic type $\mathcal{D}$ is equipped with a binary associative operator $+$ with a neutral element 0. Then, for every tiled $\mathcal{D} \times \mathcal{D}$-stream t, we define the tiled $\mathcal{D}$-stream $merge(t)$ by $d(merge(t)) = d(t)$ and, for every $k \in \mathbb{Z}$, by

$$merge(t)(k) = \pi_1(t)(k) + \pi_2(t)(k)$$

where $\pi_1(t) : \mathbb{Z} \to \mathcal{D}$ (resp. $\pi_2(t) : \mathbb{Z} \to \mathcal{D}$) is the first projection (resp. the second projection) of t, that is, the tiled $\mathcal{D}$-stream with the same synchronization length as t and such that $t(k) = (\pi_1(t)(k), \pi_2(t)(k))$ for every $k \in \mathbb{Z}$.

For strings, let φ be the mapping that maps every $\mathcal{D}$-string m to the tiled $\mathcal{D}$-stream $\varphi(m)$ defined by : $d(\varphi(m)) = |m|$ and, for every $k \in \mathbb{Z}$, defined by $\varphi(m)(k) = m(k)$ when $0 \leq k < |m|$ and by $\varphi(m)(k) = 0_{\mathcal{D}}$ otherwise.

Lemma 3.8 (Strings embeddings) *The mapping φ from $\mathcal{D}$-strings to tiled $\mathcal{D}$-streams is a one-to-one homomorphism that maps string concatenation to (merged) tiled product, i.e. for every $\mathcal{D}$-strings m_1 and m_2 we have*

$$\varphi(m_1 \cdot m_2) = merge(\varphi(m_1) \,;\varphi(m_2))$$

For streams, let ψ be the mapping that maps every $\mathcal{D}$-stream s to the tiled $\mathcal{D}$-stream $\psi(s)$ defined by $d(\varphi(s)) = 0$ and, for every $k \in \mathbb{Z}$, by $\psi(m))(k) = s(k)$ when $0 \leq k$ and by $\psi(m))(k) = 0_{\mathcal{D}}$ otherwise.

Lemma 3.9 (Streams embeddings) *The mapping ψ from $\mathcal{D}$-streams to tiled $\mathcal{D}$-streams is a one-to-one homomorphism that maps stream parallel product to tiled product, i.e. for every $\mathcal{D}$-stream s_1 and s_2 we have*

$$\psi(s_1|s_2) = \psi(s_1) \,;\psi(s_2)$$

In particular, when restricted to tiled streams with zero synchronization length, the tiled stream product is, up to isomorphism, commutative.

We also observe that these embeddings of strings and streams into tiled streams preserve mixed products as well.

Lemma 3.10 (Mixed embeddings) *For every $\mathcal{D}$-string m and $\mathcal{D}$-stream s, we have*

$$\psi(m :: s) = R(merge(\varphi(m) \,;\psi(s)))$$

4 Static (out of time) T-calculus

In the previous section, we have defined tiled streams and the related notion of tiled product. We have illustrated their relevance for being used in music modeling.

We aim now at defining an associated programming languages that allows for generalizing this approach to tiled streams of arbitrary type with a buit-in uniform way of lifting every function on elements of a given type to a function on tiled streams built over the same type, tiled stream product being a particular cas of such a lift.

We present in this section a static version of the T-calculus, that is, the T-calculus with no I/O mechanisms. Effectivity issues, that is, how to compute and run T-programs in real time, are adressed in Section 5.

4.1 Syntax

Basic constants and types. We assume that there is some set of basic types such as booleans: `bool`, positive integers: `natural`, or sets of events: `eventSet(E)` for some (finite) predefined set of events `E`, etc.

We assume, in a way somehow related with the semantics of the *none* construct in [7], that every basic type $\mathcal{D}$ we use is equipped with a sum $+$ and a neutral element $0 \in \mathcal{D}$ for that sum, i.e. such that $x + 0 = 0 + x = 0$ for every x of type $\mathcal{D}$.

Though this is not a general necessity, the above types are also and even uniformly seen as *semiring* structures with (infix) associative sum `+`, e.g. union for event sets, (infix) associative product `*` that distributes over sum, e.g. intersection for event sets, with a zero `0` that is neutral for sum and absorbant for product, e.g. empty set for event sets, and a unit `1` that is neutral for product, e.g. the set `E` of all events for event sets.

They are associated with constants. In particular, we use the notation `{a,b,c}` for the set of three events `a`, `b` and `c` $\in$ `E`. To keep notations simple, we drop any subscript (or any other mark) that may refer to the type of these constants and operators though they are implicitly considered as unambiguously typed.

Programs. A T-calculus program p is just a term built by means of the program constructs described in Figure 13 where `c` is a constant, `x` is any variable, `f` is any n-ary function symbol, $\top$ any binary operator, `;` is the synchronization product operator already presented in the previous sections, and where p_1, p_2, $\ldots$, p_i are any syntactically simpler programs. The last two constructs, operators and synchronization product, derive from the others and will thus be treated as such.

As a matter of simplification, we define no notion of variable scope. Thus we also assume that:

$p ::=$		– *primitive constructs* –
	`c`	(constant)
$\mid$	`x`	(variable)
$\mid$	`f`$(p_1, p_2, \cdots, p_n)$	(function lifting)
$\mid$	`x =` p_1	(declaration)
$\mid$	`R(`p_1`)`	(sync. reset)
$\mid$	`L(`p_1`)`	(sync. co-reset)
		– *derived constructs* –
$\mid$	p_1 `op` p_2	(operator)
$\mid$	p_1 `;` p_2	(synchronization product)

Figure 13: Static T-calculus syntax

▷ *For every program p, for every variable* `x` *occurring in p, there is at most one subprogram of p of the form* `x =` $p_{\mathtt{x}}$ *with* $p_{\mathtt{x}}$ *from now on called the* definition of `x` *in the program p.*

4.2 Static synchronization types

Every T-calculus program will be interpreted as a tiled $\mathcal{D}$-stream, also called an α-stream, when α is the type of elements of $\mathcal{D}$. As there shall be no surprise with the typing of the elements of tiled streams, we concentrate on the typing of the tiled streams resulting from T-calculus programs.

More precisely, for every T-program p, we define the typing relation

$$\Gamma \vdash p : (d, \alpha)$$

that means *the program p in environment Γ is a tiled α-stream with synchronization length d.* The typing environment Γ tells what the types of the variables that occur in p are. It is represented as a set of pairs of the form $(\mathtt{x}, (d, \alpha))$. We also write α_c for the basic type of the constant `c`. This implicitly means that every constant is written in such a way that its type is unambiguous. However, in all examples and rules given below, we keep the notation simple with constant 0 meaning the constant 0_α for any of the basic type α. The typing relation is inductively defined by the typing rules in Figure 15.
Additionally, we provide in Figure 16 the derived rules associated with the last two program constructs.

Remark 4.1 As one may have expected, the typing rule for a binary infix operator `op` derives from the mapping rule in Figure 15. The typing rule for the synchronization product p_1 `;` p_2 also derives from that same rule when applied

▷ Constants:
$$\frac{}{\Gamma \vdash \mathtt{c} : (1, \alpha_c)}$$

▷ Variables:
$$\frac{(\mathtt{x}, (d, \alpha)) \in \Gamma}{\Gamma \vdash \mathtt{x} : (d, \alpha)}$$

▷ Mapping:
$$\frac{\Gamma \vdash p_i : (d_i, \alpha_i) \quad (i \in [1, n])}{\Gamma \vdash \mathtt{f(}p_1, \cdots, p_n\mathtt{)} : (d_1 + \cdots + d_n, \alpha)}$$
with $\mathtt{f} : \alpha_1 \times \alpha_2 \times \cdots \times \alpha_n \to \alpha$

▷ Declaration:
$$\frac{\Gamma \vdash \mathtt{x} : (d, \alpha) \quad \Gamma \vdash p : (d, \alpha)}{\Gamma \vdash \mathtt{x = } p : (d, \alpha)}$$

▷ Sync. reset:
$$\frac{\Gamma \vdash p : (d, \alpha)}{\Gamma \vdash \mathtt{R(}p\mathtt{)} : (0, \alpha)}$$

▷ Sync. co-reset:
$$\frac{\Gamma \vdash p : (d, \alpha)}{\Gamma \vdash \mathtt{L(}p\mathtt{)} : (0, \alpha)}$$

Figure 14: Static type rules

▷ Operator:
$$\frac{\Gamma \vdash p_1 : (d_1, \alpha_1) \quad \Gamma \vdash p_2 : (d_2, \alpha_2)}{\Gamma \vdash p_1 \,\mathtt{op}\, p_2 : (d_1 + d_2, \alpha_3)}$$
with $\mathtt{op} : \alpha_1 \times \alpha_2 \to \alpha_3$

▷ Sync. product:
$$\frac{\Gamma \vdash p_1 : (d_1, \alpha_1) \quad \Gamma \vdash p_2 : (d_2, \alpha_1)}{\Gamma \vdash p_1 \,\mathtt{;}\, p_2 : (d_1 + d_2, \alpha_1 \times \alpha_2)}$$

Figure 15: Derived static type rules

to the identity mapping $\mathtt{id}_{\alpha_1 \times \alpha_1} : \alpha_1 \times \alpha_1 \to (\alpha_1 \times \alpha_1)$, one per types α_1 and α_2, hence leading to the derived rule described in Figure 16.

Theorem 4.2 *For every program p, for every environment Γ, there is at most one type (d, α) such that $\Gamma \vdash p : (d, \alpha)$. Moreover, it is decidable if there exists Γ and (d, α) such that $\Gamma \vdash p : (d, \alpha)$.*

Proof. Deciding of the existence of the basic type α is standard. It thus poses no difficulty. One may even imagine to allow polymorphic types as in languages like ML or Haskell. Provided Γ is given, uniqueness follows from a bottom up application of typing rules on programs.

Deciding of the existence of synchronization length typing easily reduces to

the resolution, on positive or null integers, of a finite system of linear fixpoint[1] equations.

More precisely, given a program p and an enumeration $\{p_i\}_{i\in I}$ of its sub-terms, writing $\{z_i\}_{i\in I}$ for the synchronization length of the sub-term p_i, the inference rules induce a system of equations of the form

$$z_i = \sum_{j\in I} a_{i,j} z_j + b_i$$

on per $i \in I$, with integer $a_{i,j} \geq 0$ for every $j \in I$ and $b_i \geq 0$. These equations are then solved by Gaussian elimination with simplification of the following form: if $z_i \neq 0$ above then either $z_i = 1$ and we necessarily have $0 = \sum_{j\neq i} a_j z_{i,j} + b_i$ which entails further simplifications, or $z_i > 1$ and the system has no solution. □

Example 4.3 The program `x1 = c + R(x1)` can be typed with synchronization length 1 since $d(c) = 1$. The program `x2 = L(x2) + c + R(x2)` can be typed similarly.

On the contrary, neither the program `x3 = c + x3` nor the program `x4 = x4 + c + x4` can be typed for they would have a right- or bi-infinite synchronization interval.

4.3 Semantics

Let p be a program. Let $\mathcal{X}_p$ be the set of variables that occur in p. Let Γ be a type assignment of variables such that $\Gamma \vdash p : (d, \alpha)$.

A valuation $\mathcal{E}$ for p is a map that associates every variable $\mathtt{x} \in \mathcal{X}_p$ of p with a tiled stream $\mathcal{E}(\mathtt{x})$. It is coherent with Γ when, for every variable $\mathtt{x} \in \mathcal{X}_p$, if $(\mathtt{x}, (d_\mathtt{x}, \alpha_\mathtt{x})) \in \Gamma$ then the tiled stream $\mathcal{E}(\mathtt{x})$ is a tiled $\alpha_\mathtt{x}$-stream with synchronization length $d_\mathtt{x}$.

A semantics for the program p under the valuation $\mathcal{E}$, assumed to be coherent with Γ, is then a mapping $[\![]\!]_\mathcal{E}$ that maps every subprogram q of the program p to a tiled stream $[\![q]\!]_\mathcal{E}$ such that the fixpoint property (Y) described in Figure 17 is satisfied – where, as already mentioned, $p_\mathtt{x}$ is the unique term such that $\mathtt{x} = p_\mathtt{x}$ occurs in the program, as well as the rules described in Figure 18.

(Y) *For every* $\mathtt{x} \in \mathcal{X}_p$ *we have* $\mathcal{E}(\mathtt{x}) = [\![p_\mathtt{x}]\!]_\mathcal{E}$.

Figure 16: Declaration soundness rule

Remark 4.4 As already announced in the introduction, operations on tiled streams are indeed defined as synchronizations followed by fusions. This becomes especially clear in the (tiled stream) lifting of a mapping $\mathtt{f} : \alpha_1 \times \alpha_2 \to \alpha_3$

[1] thanks to the variable declaration rule

▷ Constants: $d([\![\mathtt{c}]\!]_{\mathcal{E}}) = 0$ and

$$[\![\mathtt{c}]\!]_{\mathcal{E}}(k) = \begin{cases} \mathtt{c} & \text{when } k = 0, \\ 0 & \text{when } k \neq 0, \end{cases}$$

▷ Variable: $[\![\mathtt{x}]\!]_{\mathcal{E}}(k) = \mathcal{E}(\mathtt{x})(k)$,

▷ Mapping: $d([\![\mathtt{f}\,(p_1, \cdots, p_n)]\!]_{\mathcal{E}}) = \sum_{i\in[1,n]} d([\![p_i]\!]_{\mathcal{E}})$

$$\text{and } [\![\mathtt{f}\,(p_1, \cdots, p_n)]\!]_{\mathcal{E}}(k) = \mathtt{f}(v_1, \cdots, v_n)$$

with $v_i = [\![p_i]\!]_{\mathcal{E}}\left(k - \sum_{1\leq j<i} d([\![p_j]\!]_{\mathcal{E}})\right)$,

▷ Declaration: $d([\![\mathtt{x}\,\mathtt{=}\,p_{\mathtt{x}}]\!]_{\mathcal{E}}) = d([\![p_{\mathtt{x}}]\!]_{\mathcal{E}})$ and

$$[\![\mathtt{x}\,\mathtt{=}\,p_{\mathtt{x}}]\!]_{\mathcal{E}}(k) = [\![p_{\mathtt{x}}]\!]_{\mathcal{E}}(k)$$

▷ Sync. reset: $d([\![\mathtt{R}\,(p_1)]\!]_{\mathcal{E}}) = 0$ and

$$[\![\mathtt{R}\,(p_1)]\!]_{\mathcal{E}}(k) = [\![p_1]\!]_{\mathcal{E}}(k)$$

▷ Sync. co-reset: $d([\![\mathtt{L}\,(p_1)]\!]_{\mathcal{E}}) = 0$ and

$$[\![\mathtt{L}\,(p_1)]\!]_{\mathcal{E}}(k) = [\![p_1]\!]_{\mathcal{E}}(k + d([\![p_1]\!]_{\mathcal{E}}))$$

for every $k \in \mathbb{Z}$.

Figure 17: Static semantics rules

to a program construct of the form $\mathtt{f}\,(p_1, p_2)$ on some tiled α_1-stream p_1 and some tiled α_2-stream p_2. Indeed, computing $\mathtt{f}\,(p_1, p_2)$ amounts to:

- ▷ Synchronization (in time): computing the synchronized product $p_1\ \mathtt{;}\ p_2$ on the tiled α_1- and α_2-streams passed as arguments,
- ▷ Fusion (in space): applying mapping $\mathtt{f}$ in a point wise fashion on the resulting tiled $\alpha_1 \times \alpha_2$-stream in order to build the expected α_3-stream.

This feature already appeared in the typing rules in Figure 15. It has been made explicit in Figure 18.

The semantics of infix operators and synchronized products just derives from the primitive semantics rules stated in Figure 18. The corresponding rules are stated in Figure 19.

▷ Operator: $d([\![p_1 \,\mathtt{op}\, p_2]\!]_\mathcal{E}) = d([\![p_1]\!]_\mathcal{E}) + d([\![p_2]\!]_\mathcal{E})$ and

$$[\![p_1 \,\mathtt{op}\, p_2]\!]_\mathcal{E}(k) = [\![p_1]\!]_\mathcal{E}(k) \,\mathtt{op}\, [\![p_2]\!]_\mathcal{E}(k - d([\![p_1]\!]_\mathcal{E}))$$

▷ Synchronization product: $d([\![p_1 \,;\, p_2]\!]_\mathcal{E}) = d([\![p_1]\!]_\mathcal{E}) + d([\![p_2]\!]_\mathcal{E})$ and

$$[\![p_1 \,;\, p_2]\!]_\mathcal{E}(k) = ([\![p_1]\!]_\mathcal{E}(k), [\![p_2]\!]_\mathcal{E}(k - d([\![p_1]\!]_\mathcal{E})))$$

for every $k \in \mathbb{Z}$.

Figure 18: Derived static semantics rules

Example 4.5 Continuing Example 4.3, the tiled stream associated to `x1` is uniquely defined. It equals `c` when $k \geq 0$ and equals `0` when $k < 0$. The tiled stream associated to `x2` is also uniquely defined and equals `c` everywhere.

As another example, with uniquely defined semantics, the program `x3 = L(x3) + 0 + 1 + R(x3)` evaluates into a tiled stream of alternating `0` and `1`.

On the contrary, a program like `x = x` has many possible semantics, one per valuation of `x` although it can be typed. A program like `x = R(2 + x)`, that can also be typed, has no semantics since its value on 0 shall be infinite.

We easily check that:

Lemma 4.6 *When $\mathcal{E}$ is coherent with Γ then, for every subprogram p_1 that occurs in p, if $\Gamma \vdash p_1 : (p_1, \alpha_1)$ then $[\![p_1]\!]_\mathcal{E}$ is a tiled α_1-stream with $d([\![p_1]\!]_\mathcal{E}) = d_1$.*

4.4 Iterative semantics

Since a program may have zero, one or several semantics as illustrated in Example 4.5, we privilege below a static iterative semantics that, when defined, provides a unique semantics.

The distance between two tiled streams s_1 and s_2 of the same type is defined to be $d(s_1, s_2) = 1$ when $d(s_1) \neq d(s_2)$ and, when $d(s_1) = d(s_2)$, to be either $d(s_1, s_2) = 0$ when $s_1 = s_2$ or to be $d(s_1, s_2) = 1/2^n$ otherwise where n is the greatest integer such that, for all $-n < k < n$ we have $s_1(k) = s_2(k)$. This metric is extended to environments by letting $d(\mathcal{E}_1, \mathcal{E}_2) = \max\{d([\![\mathtt{x}]\!]_{\mathcal{E}_1}, [\![\mathtt{x}]\!]_{\mathcal{E}_2}) : x \in \mathcal{X}_p\}$. The resulting topology is called the prefix topology.

Given a typed program p, let $\mathcal{E}_{p,0}$ be the valuation that maps every variable `x` that occurs in p to the constant tiled stream 0. For every $n \in \mathbb{N}$, let then $\mathcal{E}_{p,n+1}$ be the valuation defined, for every variable $\mathtt{x} \in \mathcal{X}_p$, by $\mathcal{E}_{p,n+1}(\mathtt{x}) = [\![p_\mathtt{x}]\!]_{\mathcal{E}_{p,n}}$ for the definition $p_\mathtt{x}$ of `x` in p, where $[\![p_\mathtt{x}]\!]_{\mathcal{E}_k}$ is computed following the rules of Figure 18.

Theorem 4.7 *When $\{\mathcal{E}_{p,k}\}_{k\in\mathbb{N}}$ converges towards a limit $\mathcal{E}_p$ then, for every $\mathtt{x} \in \mathcal{X}_p$ we have $[\![p]\!]_{\mathcal{E}_p} = \mathcal{E}_p(\mathtt{x})$ and $[\![]\!]_{\mathcal{E}_P}$ satisfies the semantics rules of FIgure 18.*

In that case, we thus say that $[\![p]\!]_{\mathcal{E}_p}$ is the iterative semantics of p. We aim now at finding simple conditions that guarantee its existence and its computability in a real time fashion.

Remark 4.8 In our application perspectives as in the proposed examples, it makes sense to define 0 as the default value for it corresponds to silence.

4.5 Out of time musical examples

Let `E` be a set of MIDI-like musical event defined as follows. For every note `N`, there is an event `N` for the *noteOn* event for that note and an event `Nc` for its continuation event *noteCont*. Of course, as in MIDI, by note we mean a pitch class but possibly extended with a track number, an instrument number, an energy level, etc. The main interest of such a variant is that the empty event set 0 denotes silence. Then, as shown in the following examples, melodic lines can easily be encoded by tiled streams of type `eventSet(E)`. In these examples, operator $*$ denotes the (tiled extension of the) intersection of event sets and operator $+$ denotes the (tiled extension of the) union of event sets.

Example 4.9 (Simple melodies) We assume that the elementary durations of events are eighth notes. Writing `<N:d>` for every note `N` for the program `{N}+{Nc}+`$\cdots$`+{Nc}` when `Nc` is repeated $d-1$ times, then the tiled stream

```
a = <C4:2>
```

defines the note `C4` with a duration of two eighths, that is a quarter. In that case, the synchronization interval of that program correspond to the interval when the note is played.

This notation easily extends to event sets, hence $<$ `0 : 6` $>$ denotes a rest of duration 6 eighths (and same synchronization duration). Then, the tiled stream

```
b = L(B3) + <C4:2> + <0:6>
```

simply encodes a 4 beats measure composed of one quarter note `C4` followed by silences, that is preceded by an anacrusis made of an eighth note `B3`.

Such a fairly simple example clearly shows how all finite examples that are given in Section 2 can be encoded in our proposal. The binary function *merge* that is used in Section 3 is simply defined by $merge(p_1; p_2) = p_1 + p_2$ for every tiled stream p_1 and p_2.

Example 4.10 (An infinite canon) Another example is the infinite canon one can be built as follows. Given four melodic lines `m1`, `m2`, `m3` and `m4` built with the techniques shown above assumed to be of the same (non zero) synchronization length. Let us consider the program defined by

```
y = (x = m1+R(m2+m3+m4+x))
    + x + x + x + R(y)
```

In that construction, using the reset operator on `m2+m3+m4+x` ensures that `x` has the same synchronization as `m1`. Then, for a similar reason, the synchronization length of `y` is four times that of `m1` and thus, since all melodic lines have the same synchronization length, this canon can indeed be depicted, before pointwise summation, as in Figure 20. Observe that such an encoding even allows

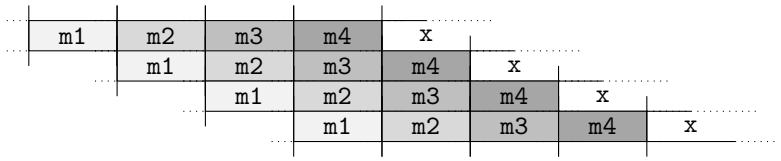

Figure 19: The canon before merge

for canons built on melodic lines with anacrusis and/or late conclusive notes or sequences.

Example 4.11 (Local stream processing) Last example, we show how local stream processing function can be encoded in our calculus. Let

$$F : (\mathbb{Z} \to \alpha_1) \to (\mathbb{Z} \to \alpha_2)$$

be a (bi-infinite) stream processing function assumed to be locally computable, i.e. there is m and $n \in \mathbb{N}$ with $m \geq n$ and a mapping $g : \alpha_1^{n+1} \to \alpha_2$, such that, for every input stream $s : \mathbb{Z} \to \alpha_1$, for every $k \in \mathbb{Z}$,

$$F(s)(k+m) = g(s(k), s(k+1), \cdots s(k+n))$$

i.e. $F(s)$ is computed with delay m from a sliding window on s of length $n+1$. Then F can be encoded by the program defined as follows:

```
<0:m> + R(g(<0:0> + s;<0:1>+s;
           <0:2>+s;...;<0:n>+s))
```

In that encoding, we choose to make the synchronization length of the resulting tiled stream to be equal to `m`.

5 Dynamic (in-time) T-calculus

We aim now at analyzing more in depth our programming language proposal so that it can handle outputs and inputs in real time.

Handling outputs in real time means first that a program need to be started at some point, hence there must be a computable (relative) starting date before which all values of the tiled stream computed by the program are known to be 0 (silent), i.e. the program (and its subprograms) must have a *finite past*.

Then, all processing must be *finite* and *causal*, that is, every output value must only depends on current or finitely many past computed values that have been memorized.

Handling inputs in real time means extracting on the fly, from real time input streams (with absolute dates), some tiled streams (with relative dates) that can be used at other place as partial echos of the recorded inputs.

For developing such analysis tools, we assume from now on that the program p is well typed under some typing environment Γ. For the time being, we also assume that the program p has no input, that is, every variable $\mathtt{x} \in \mathcal{X}_p$ that occurs in p has a (unique) definition $p_{\mathtt{x}}$ in p. The case of input is handled in Section 5.4 below.

5.1 Synchronization profiles

We first aim at defining a type system that computes a possible realization interval for the program p (see Definition 3.2) compatible with its iterative semantics (see Section 4.4) in the case it admits one. This will induce a sufficient condition for the program p to have a finite past.

Definition 5.1 (Synchronization profile) *Let $\bar{\mathbb{N}}$ be the set of zero or positive integers extended with a greatest element ∞. A synchronization profile for the program p is a triple $(l, d, r) \in \bar{\mathbb{N}} \times \mathbb{N} \times \bar{\mathbb{N}}$ such that d is the synchronization length of program p, i.e. $\Gamma \vdash (d, \alpha)$ for some type α, and such that, for every $k \in \mathbb{Z}$, if $[\![p]\!]_{\mathcal{E}}(k) \neq 0$ then $-l \leq k \leq d + r$, with $-l \leq k$ (resp. $k \leq d + r$) that is true whenever $l = \infty$ (resp. $r = \infty$), i.e. interval $[-l, d + r] \subseteq \mathbb{Z}$ is a realization interval of the program p semantics.*

Before giving a set of rules to compute synchronization profile types, the notion of synchronization profile itself, and the way it evolves in a sum, is depicted in Figure 21.

Remark 5.2 The product of triples $(l, d, r) \cdot (l', d', r')$ defined to be

$$(\max(l, l' - d), d + d', \max(r - d', r'))$$

as depicted in Figure 21, is known to induce over the set $\mathbb{N} \times \mathbb{N} \times \mathbb{N}$ a monoid structure, i.e. the product is associative and it has $(0, 0, 0)$ as neutral element.

Moreover, this monoid is isomorphic to the submonoid of positive elements of the free inverse monoid of one generator (see e.g. [17]). The synchronization profiles, that also allows ∞ as left or right value, form a submonoid of the filter completion of that monoid.

Generalizing this observation to all program constructs, we obtain the set of rules depicted in Figure 22 where Δ is an environment that associates every variable to its synchronization profile.

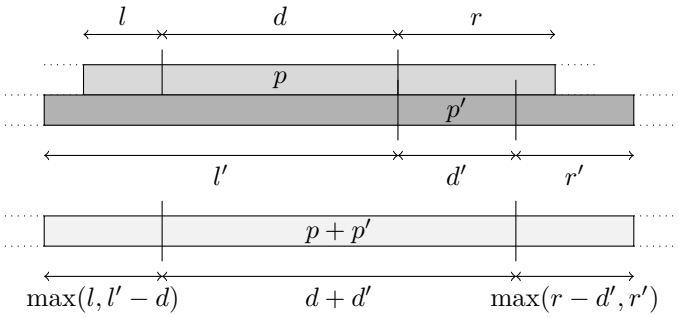

Figure 20: The synchronization profile of a tiled sum

We say that a profile (l, d, r) is smaller than a profile (l', d', r') when $l \leq l'$, $d = d'$ and $r \leq r'$. One can easily check that this is an order relation[2]. It is extended point-wise to synchronization profile environment.

Lemma 5.3 *The least synchronization profiles (and the corresponding environment Δ) satisfying the rules describe in Figure 22 is computable. Moreover, when $\Delta \vdash p : (l, d, r)$ with finite l, we know that for every $k \leq -l$ we have $[\![p]\!]_{\mathcal{E}} = 0$, that is, the program p is* finite in the past*.*

Proof. The computability of synchronization lengths follows from the fact that we assume that p is typed in a given environment Γ.

Computability of the left and right parts of a synchronization profile follows from the fact that the rules depicted in Figure 22 induce conjunctions of inequalities of the form

$$l_i \geq c_i + \sum_{j \in J} l_j$$

which, whenever $c_i \neq 0$ and $i \in J$, implies that $l_i = \infty$. That fact allows for accelerating the least solution computation that otherwise may be infinite. □

For instance, the program `x = 1 + R(x) + 1` has the minimum synchronization profile $(0, 1, \infty)$.

5.2 Direct temporal dependencies

We aim now at finding simple (computable) sufficient conditions under which the iterative semantics of the program p exists and can be computed in a causal way.

Before providing a general solution, let us examine some examples.

[2]It is even the reverse order of the natural order of the underlying inverse monoid.

▷ Constants:
$$\frac{}{\Delta \vdash \mathtt{c} : (0,1,0)}$$

▷ Variables:
$$\frac{(\mathtt{x},(l,d,r)) \in \Delta}{\Delta \vdash \mathtt{x} : (l,d,r)}$$

▷ Mapping:
$$\frac{\Delta \vdash p_i : (l_i,d_i,r_i) \quad (i \in [1,n])}{\Delta \vdash \mathtt{f}(p_1,\cdots,p_n) : (l,d,r)}$$
with $l = \max\left(l_i - \sum_{1\leq j<i} d_j\right)$, $d = \sum_i d_i$,
and $r = \max\left(r_i - \sum_{i<j\leq n} d_j\right)$,

▷ Declaration:
$$\frac{\Delta \vdash \mathtt{x} : (l,d,r) \quad \Delta \vdash p : (l,d,r)}{\Delta \vdash \mathtt{x}=p : (l,d,r)}$$

▷ Sync. reset:
$$\frac{\Delta \vdash p : (l,d,r)}{\Delta \vdash \mathtt{R}(p) : (l,0,d+r)}$$

▷ Sync. co-reset:
$$\frac{\Delta \vdash p : (l,d,r)}{\Delta \vdash \mathtt{L}(p) : (l+d,0,r)}$$

Figure 21: Synchronization profile type rules

Example 5.4 Given a constant tile $\mathtt{m}$ with a non zero synchronization length d, assume that p is a program of the form $\mathtt{x} = \mathtt{m} + \mathtt{R}(\mathtt{x})$. Then the program p is causal since the value of $\mathtt{x}$ at any date k only depends on constant values from $\mathtt{m}$ and on the value of $\mathtt{x}$ at instant $k - d$.

If the program p is of the form $\mathtt{x} = \mathtt{L}(\mathtt{x}) + \mathtt{m}$ then, on the contrary, it is not causal since the value of $\mathtt{x}$ at some date k may depends on the value of $\mathtt{x}$ at instant $k + d$.

These examples show that we need to analyse the temporal dependencies between objects that are handled by a program. This is done in two steps, first by defining direct temporal dependencies and then by defining iterated temporal dependencies.

Definition 5.5 (Direct temporal dependency mapping) *Let δ be the mapping that maps every pair of subprogram q of p and every variable $\mathtt{x} \in \mathcal{X}_p$ to the set $\delta(q,\mathtt{x}) \subseteq \mathbb{Z}$ that is inductively defined by the rules in Figure 23, where we denote $d(q) \in \mathbb{N}$ the synchronization length of any subprogram q and, for any set $X \subseteq \mathbb{Z}$ and any $n \in \mathbb{Z}$, we denote by $X - n$ the set $\{x - n \in \mathbb{Z} : x \in X\}$.*

The mapping δ is clearly computable in finite time by a bottom up traversal of the syntactic structure of the program p and only produces finite subsets of $\mathbb{Z}$.

Example 5.6 Let p be the program `x= 1 + R(1 + L(x))` that is depicted in

▷ Constants: $\delta(\mathtt{c},\mathtt{x}) = \emptyset$,

▷ Variable: $\delta(\mathtt{y},\mathtt{x}) = \emptyset$ when $\mathtt{x}$ and $\mathtt{y}$ are distinct
and $\delta(\mathtt{x},\mathtt{x}) = \{0\}$ otherwise,

▷ Mapping: $\delta(\mathtt{f}(p_1,\cdots,p_n),\mathtt{x}) =$

$$\bigcup_{1\leq i\leq n}\left(\delta(p_i,\mathtt{x}) - \sum_{1\leq j<i} d(p_j)\right),$$

▷ Declaration: $\delta(\mathtt{y}=p_{\mathtt{y}},\mathtt{x}) = \delta(\mathtt{y},\mathtt{x})$,

▷ Sync. reset: $\delta(\mathtt{R}(p_1),\mathtt{x}) = \delta(p_1,\mathtt{x})$,

▷ Sync. co-reset: $\delta(\mathtt{L}(p_1),\mathtt{x}) = \delta(p_1,\mathtt{x}) + d(p_1)$.

Figure 22: Direct temporal dependencies rules

Figure 24, where, for the sake of simplicity we have omitted to picture the reset operator. The computation of δ can then be illustrated as follows.

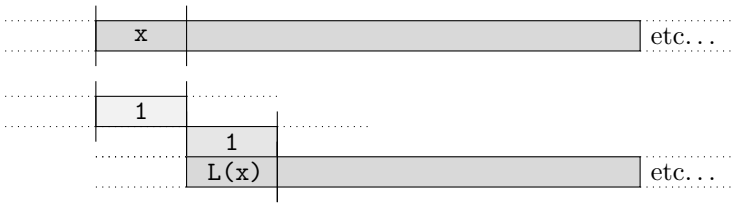

Figure 23: Observing temporal dependencies

We have $\delta(\mathtt{1},\mathtt{x}) = \emptyset$ and $\delta(\mathtt{x},\mathtt{x}) = \{0\}$. Since $d(x) = 1$, by applying the co-reset rule, we thus have $\delta(\mathtt{L(x)},\mathtt{x}) = \{1\}$. By applying the mapping rule, we have $\delta(\mathtt{1 + L(x)},\mathtt{x}) = \{0\}$ hence, by applying the reset rule, $\delta(\mathtt{R(1 + L(x))},\mathtt{x}) = \{0\}$ and thus

$$\delta(\underbrace{\mathtt{1 + R(1 + L(x))}}_{p_{\mathtt{x}}},\mathtt{x}) = \{-1\}$$

We easily check that the iterative semantics of p is well defined with, for every $k \in \mathbb{Z}$,

$$[\![x]\!]_{\mathcal{E}}(k) = \begin{cases} 0 & \text{if } k < 0 \\ [\![x]\!]_{\mathcal{E}}(k-1) + 1 & \text{if } k \geq 0. \end{cases}$$

In other words, on such a simple example, the direct temporal dependency mapping δ tells us how $\mathtt{x}$ depends on itself.

In the more general case, we have:

Lemma 5.7 *Let $\mathcal{E}$ be a valuation of the variables of $\mathcal{X}_p$ compatible with the typing environment and satisfying the fixpoint property, i.e. $\mathcal{E}(\mathtt{x}) = [\![p_{\mathtt{x}}]\!]_{\mathcal{E}}$ for every $\mathtt{x} \in \mathcal{X}_p$. Then, for every subprogram q of p, there is a function*

$$f_q : \prod\{\alpha_{\mathtt{y}} : \mathtt{y} \in \mathcal{X}_q, d \in \delta(q, \mathtt{y})\} \to \alpha_q$$

with α_q (resp. $\alpha_{\mathtt{y}}$) being the basic type of q (resp. the basic type of $\mathtt{y}$) such that, for every $k \in \mathbb{Z}$,

$$[\![q]\!]_{\mathcal{E}}(k) = f_q\left(\mathcal{E}(\mathtt{y})(k+d) : \mathtt{y} \in \mathcal{X}_q, d \in \delta(q, \mathtt{y})\right)$$

Proof. Immediate from the definition of mapping δ. Indeed, direct temporal dependencies are invariant under translation, and the rules stated in Figure 23 compute them for $k = 0$. □

5.3 Iterated temporal dependencies

The previous lemma yet does not say a word on the (causal) computability of the program p. Indeed, nested or even mutually recursive definitions of variables may occur in the program p.

Definition 5.8 (Iterated temporal dependency mapping) *Let δ^* be the mapping that maps every pair of subprogram p and variable $\mathtt{x}$ to the set $\delta(q, \mathtt{x}) \subseteq \mathbb{Z}$ that is inductively defined as indicated in Figure 25 where, for every two sets*

$$\delta^*(q, \mathtt{x}) = \delta(q, \mathtt{x}) \cup \textstyle\bigcup_{\mathtt{y} \in \mathcal{X}_p} \left(\delta(q, \mathtt{y}) + \delta^*(p_{\mathtt{y}}, \mathtt{x})\right)$$

Figure 24: Iterated temporal dependencies

of integer X and $Y \subseteq \mathbb{Z}$, we write $X + Y$ for the set $X + Y = \{x + y \in \mathbb{Z} : x \in X, y \in Y\}$.

The mapping δ^* is also computable though it produces linear subsets of $\mathbb{Z}$, that is, finite unions of sets of the form $\{k_1 + n * k_2 : n \in \mathbb{N}\}$ with k_1 and $k_2 \in \mathbb{Z}$.

Theorem 5.9 *Assume that for every variable $\mathtt{x} \in \mathcal{X}_p$ that occurs in p the set $\delta^*(\mathtt{x}, \mathtt{x})$ only contained strictly negative values. Then the program p admits an iterative semantics that is causal and with finite past.*

Proof. (sketch of) Lemma 5.7 induces, via mapping δ, a finite set of equations that relate, for every iteration step $n \in \mathbb{N}$ (see Section 4.4) the valuation $\mathcal{E}_{n+1}$ with the valuation $\mathcal{E}_n$.

Then, thanks to the hypothesis on δ^*, the least synchronization profile of p cannot be left infinite, hence, by Lemma 5.3, there exists an index k_0 before which every subprogram of the program p evaluates to zero.

Then, by applying again the same hypothesis, we prove by induction that there exists a strictly increasing sequence of integers $\{k_n\}_{n\in\mathbb{N}} \subseteq \mathbb{Z}^{\mathbb{N}}$ such that, for every $m \geq n$, every variable $\mathtt{x} \in \mathcal{X}_p$, every $k \leq k_n$, $\mathcal{E}_m(\mathtt{x})(k) = \mathcal{E}_n(\mathtt{x})(k)$. This proves boh the convergence and the causality of the iterative semantics.

The finiteness of the memory that is needed to compute $[\![p]\!]_{\mathcal{E}}$ follows from the finiteness (and the translation invariance) of δ that is needed to compute the semantics of the program p from the value of its variables (see Lemma 5.7). □

5.4 Monitoring inputs

We aim now at handling program inputs. An input is an infinite stream of some type α that is started at some absolute date. Since our programs only handle tiled streams, we need to see it as such.

There are many ways to convert an input stream into a tilled stream. Here, we adopt a rather conservative (or pure) way that amounts to preserving transparential referency: a given input tiled stream expression always has the same evaluation, wherever it occurs in a program.

From the syntactical point of view, an input just appears as a free variable. From the semantical point of view, following Section 3.5, we first assume that such a variable denotes a tiled α-stream with zero synchronization length. Then, the bi-infinite sequence associated to an input stream is assumed to be 0 before the (relative) date zero. This relative reference date is also assumed, in absolute time, to be equal to the real time occurrence date of the (relative) date zero of the program itself. Then, the values from and after that date are equal to the corresponding real time input values.

In other words, let $i : \mathbb{Z} \to \mathcal{D}$ be a real input stream potentially seen as a bi-infinite stream indexed by absolute dates. Let x_i be the free variable associated to that input stream in the program p. Let t_0 (resp. $k_0 \leq 0$) be the absolute date (relative date) at which the program p is started. Then, the synchronization profile of the tiled monitoring x_i of the input stream i is assumed to be $(0, 0, \infty)$ and every valuation $\mathcal{E}$ on x_i is defined, for every $k \in \mathbb{Z}$, by

$$[\![x_i]\!]_{\mathcal{E}}(k) = \begin{cases} 0 & \text{if } k < 0 \\ i(t_0 + k - k_0) & \text{if } k \geq 0 \end{cases}$$

i.e. the input stream is monitored in real time from the relative date 0 of the program.

Example 5.10 *(Tiling input streams)* Let a, b, c and d be four positive integer constants and let $\mathtt{x}$ be a input tiled stream. We want to built out of $\mathtt{x}$ a finite tiled stream $\mathtt{y}$ with synchronization profile (b, c, d) such that, under the iterative

semantics, for every $k \in \mathbb{Z}$,

$$[\![\mathtt{y}]\!]_{\mathcal{E}}(k) = \begin{cases} 0 & \text{if } k < -(a+b) \\ [\![\mathtt{x}]\!]_{\mathcal{E}}(k+a+b) & \text{if } -(a+b) \leq k \leq c+d \\ 0 & \text{if } c+d < k \end{cases}$$

In other words, we want to monitor the input `x` from its relative date from a to $a+b+c+d$, with a resulting synchronization interval $[0, c]$ for `y` that corresponds, up to translation, to the interval $[a+b, a+b+c]$ on `x`.

It occurs that this can be done by the following program

```
y = L(R(x)+<0:a+b>)
    * (L(L(0)+ <0:a>+<1:b>)
       + <1:c> + R(<1:d>+R(0)))
```

Denoting by $A(\mathtt{x})$ the first term of the product and by F the second term of the product, such a monitoring is depicted in Figure 26 where, building $A(\mathtt{x})$,

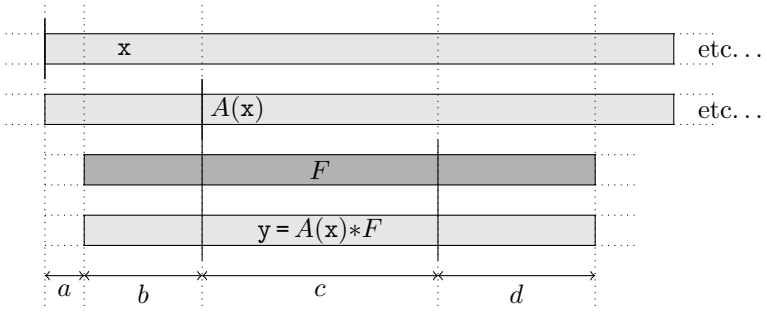

Figure 25: Tiling input `x`

we just aim at repositioning the beginning of the synchronization where needed and where F act as a boolean filter with the adequate synchronization profile.

In other words, in the T-calculus, one can sorts of monitors that dynamically create tiled streams wherever within input streams.

Remark 5.11 Most results stated above remain valid when extended with inputs. In particular, Theorem 5.9 now says that the class of causal T-calculus programs with inputs, defined by its hypothesis, captures the class of deterministic finite state sequential I/O-transducers on words (see e.g. [20]). Indeed, the converse encoding of every such a finite state transducer into a T-calculus program is easy. A single internal tiled streams suffices in order to encode the sequence of transducer states in a run.

Observe that handling inputs may create new source of incoherence. For instance, the program $\mathtt{x} + \mathtt{1} + (\mathtt{x} = \mathtt{y})$ with input `y` cannot be executed since this amounts to output the tiled stream `y` one step before it is actually input.

A simple remedy to such an erroneous usage is to require that in the program p, we only have negative or null iterated temporal dependencies in $\delta^*(p, \mathtt{y})$.

The exemple above also show a limitation of our proposal. The associated synchronization profiles and position within the input streams must be statically defined. Defining monitoring devices that *conditionally* create tiled stream out of inputs is another and delicate issue that is left for further study. Indeed, it would certainly break transparential referency for the resulting evaluation of such a monitor would certainly depend on the real time these monitor are activated.

Remark 5.12 Last, let us observe that the question raised by allowing some conditional monitoring of input is much broader than one may expect. Indeed, other approaches, rather orthogonal, also appear in music softwares like *Antescofo* [4] that are based on score followers.

Indeed, in such kind of softwares, score followers are used as intelligent input monitors so that the distance from the current position to some conditional input event can be predicted and updated at every steps. Then, thanks to some mechanism of expansion/contraction of the execution timeline, these softwares allow for defining forward synchronization mechanisms that, at first sight, may seem to break causality.

6 Related work

The idea of distinguishing effective starts and stops from logical ones implicitly appears in the programming language Loco [5] via the `pre` and `post` program constructs. As far as the authors know, there has been no follow up to this work.

More than two decades later, our proposal, that makes such an idea explicit and formal, is based on the first author's study of rhythm representations [10]. These ideas have then been developed in [1] and experimented in [14] in the context of real-time performance with (tiled) audio signals.

The present work is also inspired by both the music description language *Elody* [18] and the signal processing language *Faust* [6]. It can be seen as a proposal for a (tiled) extension of these two functional languages.

It is worth mentioning that the underlying algebraic concepts is currently studied quite in depth in the abstract setting of formal language and inverse semigroup theory [12, 13, 11]. Since (regular) languages of tiles can be interpreted as (simple) program types, we still expect more interconnections between these two research axis.

It must be added that, though orthogonal, our proposal has many connection points with the DSL Euterpea [8], and, more specifically, Hudak's proposal for temporal polymorphic media [7]. This work gives many clues for extending the present proposal to tiled streams of elements with durations: a study yet in progress.

7 Conclusion

We have described an embedding of music strings and streams into tiled streams. It is based on a tiled signal algebra that allows for a more accurate description of typical musical constructions such as *play together* or *play one after the other*.

The associated language, the T-calculus, is thus proposed as an implementation of these ideas. Of course, it is yet not a full programming language but indeed only a calculus. Extending our proposal with (dynamic) conditional synchronization duration and multi time scale handling is a clear necessity for achieving a truly modular programming language for time sensitive intermedia systems.

Acknowledgment

A preliminary version of this work benefited from numerous comments and advices from the participants of the INEDIT[3] workshop held in Bordeaux in spring 2013. The next preliminary version benefited from even more numerous and invaluable comments and criticisms from the anonymous referees of the FARM workshop. The authors wish to express deep gratitude to all of them.

References

[1] 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.

[2] L. Bigo. Symbolic representations and topological analysis of musical structures with spatial programming. In *JCAAAS*, Paris, 2010.

[3] L. Bigo, J.-L. Giavitto, and A. Spicher. Building topological spaces for musical objects. In *Mathematics and Computation in Music*, volume 6726 of *LNAI*, Paris, France, Juin 2011.

[4] A. Cont. Antescofo: Anticipatory synchronization and control of interactive parameters in computer music. In *International Computer Music Conference (ICMC)*, 2008.

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

[6] D. Fober, Y. Orlarey, and S. Letz. FAUST architectures design and OSC support. In *14th Int. Conference on Digital Audio Effects (DAFx-11)*, pages 231–216. IRCAM, 2011.

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

[3] see `http://inedit.ircam.fr/`

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

[9] P. Hudak, J. Hugues, S. Peyton Jones, and P. Wadler. A history of Haskell: Being lazy with class. In *Third ACM SIGPLAN History of Programming Languages (HOPL)*, San-Diego, 2007. ACM Press.

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

[11] 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.

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

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

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

[15] J. Kellendonk. The local structure of tilings and their integer group of coinvariants. *Comm. Math. Phys.*, 187:115–157, 1997.

[16] J. Kellendonk and M. V. Lawson. Tiling semigroups. *Journal of Algebra*, 224(1):140 – 150, 2000.

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

[18] 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.

[19] S. Letz et al. The LibAudioStream library, 2012. `http://libaudiostream.sourceforge.net/`.

[20] J. Sakarovitch. *Elements of Automata Theory*. Cambridge University Press, 2009.