Multi-scale design of interactive music systems : the libTuiles experiment

David Janin, Florent Berthaut, Myriam Desainte-Catherine

LaBRI, CNRS UMR 5800
Laboratoire Bordelais de Recherche en Informatique

Rapport de recherche RR-1471-13

Multi-scale design of interactive music systems : the libTuiles experiment

April 16, 2013

David Janin, Florent Berthaut, Myriam Desainte-Catherine
LaBRI, Université de Bordeaux, CNRS UMR 5800 351,
cours de la libération
F-33405 Talence, FRANCE

Multi-scale design of interactive music systems : the libTuiles experiment

D. Janin *, F. Berthaut, M. Desainte-Catherine
LaBRI, Université de Bordeaux, CNRS UMR 5800
351, cours de la libération
F-33405 Talence, FRANCE
{janin|berthaut|myriam}@labri.fr

April 16, 2013

Abstract

The design and implementation of an *interactive music system* is a difficult task. It necessitates the description of complex interplays between two design layers at least : the real time synchronous layer for audio processing, and at the symbolic event based layer for interaction handling. Tiled programming is a recent proposal that aims at combining with a single metaphor: tiled signals, the distinct programmatic features that are used in these two layers. The libTuiles experiment presented in this paper is a first experimental implementation of such a new design principle.

1 Introduction

1.1 Background

Nowadays, many specialized languages can be used for the design and implementation of musical systems. Be them textual like *Supercollider/chuck* [16] or *Faust* [7], or visual like *Max/Msp* or *PureData* [6], these languages mostly inherit from the synchronous programming language paradigms that allow for powerful descriptions of signal processing mechanisms.

However, programming interactive musical systems remains a delicate task. In particular, maintaining the time/rhythmic coherence of musical systems govern by the unpredictable arrival of asynchronous events is a task that easily becomes intractable. This can be partly explain by the heterogeneous time scales or layers at which such systems need to be described. Audio processing necessitates low level real time synchronous programming mechanisms while interaction handling necessitates high level event based system design tools.

*This work is partially supported by the french project INEDIT, ANR-12-CORD-0009.

Such a difficulty, partially adressed by the GALS design style [15], remains a challenging issue. Despite considerable effort, there is still a lack of high level metaphors or paradigms allowing for a hierarchical, multi-scale and modular description of dynamic time structuring mechanisms.

Among other proposals, the *i-score* sequencer [2] integrates an explicit specification mechanism that allows for the high level description of the relative positioning of musical objects, hence their potential *overlapping*. Together with explicit input control points and dynamic mechanisms for solving position constrains, the *i-score* sequencer thus already offers an abstract description of dynamic time structuration. However, by lack of additional control flow structures such as conditionals and loops, its applicability remains limited.

Independently, in the lines of the structuralist approach developed for musical linguistic [14], recent studies [9] emphasis the fact that, for computer assisted music systems, a key issue lays in the precise modeling of *behaviors overlaps* that recurrently occur in such (multi-agent) musical systems. Further studies, more oriented towards abstract and untimed models, provide evidences that an entire and well-developed mathematical field, *inverse semigroup theory* [13], is suitable for developing an associated language theory of *overlapping structures* [8,10,11].

1.2 Outline

The work presented here aims at combining the high level time specification mechanisms offered by the *i-score* approach with the modeling power provided by languages of overlapping structures, and with the efficient signal processing provided by the synchronous languages.

Implementing an advanced synchronization algebra of audio or musical patterns [5], the *libTuiles*, first appears as a fairly versatile multi-scale and hierarchical mixing tool. In the long run, the *libTuiles* also aims at becoming the first *execution engine* for the T-calculus [12] : the programming language theoretic counterpart of the experiment presented here.

The *libTuiles* can be connected to the real time synchronous audio thread provided by the *JACK* audio server. An additional granular synthesis module for producing audio signals that can be stretched makes it even more easy to use with tiled sound files. It is also linked with other existing tools such as the *Faust* [7] synchronous programming language.

Last but not least, a graphical interface, the *simpleTuilesLooper*, inspired by live looping interfaces such as *Drile* [4], allows for *live performance* experiments of the underlying metaphors and concepts.

2 Modeling of musical processes

Modeling musical system behaviors, be them *on time* when systems are running, or *off time* when systems are being designed, one faces the long standing and complex question of musical objects representation. Many proposals, often incomparable, are available. The *libTuiles* presented in this paper is based on a

rather formal model : *tiled signals*, that have been formalized as an attempt to clarify the situation.

2.1 The structure space of musical objects

There already exist many formalisms applicable to the modeling of music. Each of them provides answers for specific application perspectives, usage constraints and thus approximates the musical objects that are described. An immediate difficulty is therefore to understand *what* the characteristic of these models are and *which* one of their features we truly need for designing interactive music systems.

For instance, in a *western music score*, notes and rhythms are pictured in such a way that, in particular, the fast reading of melodic and rhythmic lines by musicians is made easier. In particular, bars and metric structures indicate on every system how musicians should synchronize their plays.

When modeling music for designing a music system, the visual aspect of music score is probably of a fairly low interest. However, there already appear two dimensions of some abstract modeling space where the various models of music lay. The first one, the *time axis* (T), is depicted by the sequence of notes, the succession of bars, and so on. The second one, the *parallel axis* (P), appears in the many music systems that are to be played in parallel by musicians.

Analyzing further music scores such as, for instance, popular melody annotated with chords as in jazz music scores, a third dimension appears, the *abstraction dimension* (A). Indeed, music is often described at various level of abstractions such as melodic lines, chords progressions, stylistic annotations, and so on.

Though often implicit, a fourth dimension also appears when modeling interactive (or improvised) music. It is the *interaction (or alternative) axis* (I) that allows, for instance, the descriptions of how musicians (say in a jazz band) can adapt their plays to the real time performance of a given soloist according to some stylistic rules.

In other words, music models adapted to the design of interactive music systems lay in an at least four dimensional space that is depicted in Figure 1.

Of course, such a four dimensional modeling space for musical objects is highly debatable. Even more, there may be some description of music that mix so much these dimensions that it no longer make sense to distinguish them. Still, positioning a given musical model in such a space may help clarifying our understanding of its features.

For instance, standard *piano roll* that are displayed on computer screens typically lay in the two dimensional space formed by the time axis (T) and the parallelism axis (P). Another typical example is the *musical transcription and analysis* of a recorded performance. As all possible interactions have been resolved during the performance, it lays in the three dimensional space formed by time (T), parallelism (P) and abstraction (A).

Typical models of *reactive systems* are, in computer science, *branching structures* (or input/output discrete automata) that describe, in every state, the po-

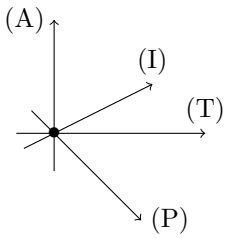

Figure 1: The four dimensional music space

tential behaviors of those systems that depend and evolve with the external events that are received. Such branching structures typically lay in the two dimensional space formed by the (abstract) time axis (T) and the interaction axis (I).

2.2 Time scale types

Designing interactive musical systems, another major source of difficulty is that the nature of modeling features may change depending on the level of abstraction. This is especially clear in the two dimensional modeling space defined by time (T) and abstraction (A) where various heterogeneous time scales can be used to describe music.

These time scales can be classified in four types at least that are described below, from the more abstract to the more concrete. Somehow worth being noticed, these types relate to the implicit time granularity they are describing: from several hours for the most abstract to few 10^-5s for the most concrete.

Logical (causal) time. Musical events are units, positioned one relatively to the other on a logical time scale which structure is described by means of modalities such as *before*, *after*, *at the same time*, etc... The time granularity goes from a few seconds, e.g. between two chords, to several minutes, e.g. between two musical movements, or even hours, between two concert's parts.

Symbolic (quantized) time. Musical objects now have abstract durations possibly measured in some symbolic units, say beats or bars, or even just atomic events. They are also positioned in the underlying symbolic time scale(s) that can remain quite abstract. More subtle modalities such as *while*, *until*, or *since* may refer to these durations. The associated time granularity is often about several seconds.

Asynchronous (placed in) real time. Musical objects are now positioned at real dates, for instance during a musical performance. Such a time positioning, that can be relevantly be heard by human hears, is measured with a precision from 10^{-1}s to 10^{-3}s. In this time scale, musical events remains irregular with, for instance, musical effect of rhythmic tension and resolution that can achieved by slightly shifting event's onsets *around* a theoretical absolute pulse position.

Synchronous (or continuous) real time. Cut into regular grains or frames, musical objects are now described as continuous (with real instruments) or digitized (with DSP) signals that refer to some regular real time clock with periods ranging from 10^{-2}s to 10^{-5}s. At that level of precision, specific techniques must be used when combining/processing musical objects in order to avoid undesirable artifacts such as, for instance, *clicks* that may be product by sudden *phase changes* at the signal level.

Designing tools for the conception of computerized music systems therefore requires to handle the modeling of all these heterogeneous time scales.

2.3 From signals to tiled signals

In every music systems, be it for mixing signals, or more generally for arbitrary multi-channel signal processing, one of the most fundamental operation consists in positioning in time, one relative to the other, the signals to be processed. This feature is depicted in Figure 2. Such an operation, that lays in the two

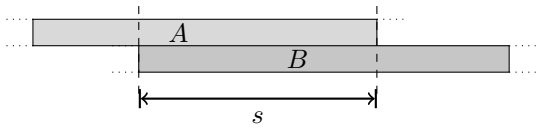

Figure 2: External synchronization

dimensional space of time (T) and parallelism (P), is often performed by means of an *external synchronization* mechanism where the relative positioning of the signals depends on the result of their combined analysis, for example relying on onset detections.

Commonly used by sound engineers in music studios, such an approach however lacks compositionality. Some audio or musical analysis may need to be performed again and again each time a new signal (or musical object) has to be positioned with respect to the previous ones. In order to avoid such a useless repeated analysis, audio processing applications are thus equipped with various and somehow adhoc notions of *time stamps* or *sync. marks* that annotate the tracks onto which these signals are positioned. It occurs that such technical tricks can be formalized with great benefits via the notion of *tiled signals*. Indeed, *tiled signals* appear when one wants to *internalize* such synchronization marks.

Simply said, a tiled signal is a signal equipped with two additional bars that delimit what are called the *synchronization window* of the tiled signal. By contrast, the position in time of the entire signal is called the *realization window*. More formally, for every signal A, the relative positioning of the synchronization window with respect to the realization window can be modeled by specifying two values : the left offset l_A and the right offset r_A, as depicted in Figure 3. With s_A the duration of the synchronization window, the resulting duration of the realization window is given by $l_A + s_A + r_A$. The resulting triple (l_A, s_A, r_A)

is called the *synchronization profile* of the tiled signal A. With this model,

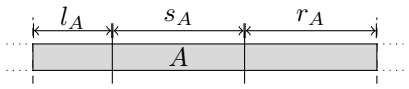

Figure 3: Synchronization vs realization windows

synchronizing two tiled signals only amounts to positioning the second bar of the first tiled signal right at the same time as the first bar of the second timed signal. This is depicted in Figure 4. The resulting synchronized product of two

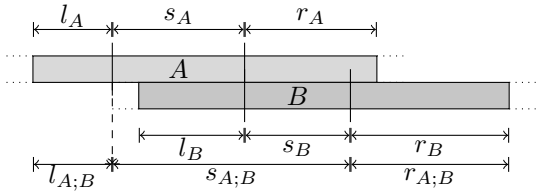

Figure 4: Internal synchronization

tiled signals A and B is denoted $SEQ(A, B)$ or simply $A; B$. An immediate observation is that the synchronization product $A; B$ of two tiled signals A and B is indeed compositional since, as depicted in Figure 4, the new built signal is again a tiled signal.

2.4 External vs internal time specifications

In an interactive sequencer such as *i-score* [2], the notion of external synchronization specification can be abstracted. More precisely, one may specify that some properties are satisfied by s, be it positive or negative. For instance, one may require that $s > 0$, meaning that the beginning of the signal B occurs before the end of signal A.

Doing so, one shifts from a *concrete* and *quantitative* specification of synchronization to an *abstract* and *qualitative* specification of signals positioning in time. The model is shifted up along the abstraction axis (A). To some extent, it even allows to combine both the realtime scale (for music performance) and symbolic time scale (for music system description) [3]. It relies on qualitative temporal logics of intervals such as Allen's logic [1].

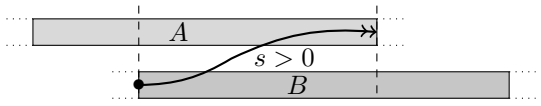

Figure 5: External synchronization constraints

It occurs that using tiled signals, such qualitative approaches, *à la Allen*, are still possible. Indeed, specifying that $s > 0$ as depicted in Figure 5 can equivalently be expressed with internal synchronization specifications as depicted in Figure 6. Indeed, this just amounts to requiring that $r_A = 0$ and $l_B \geq 0$. In

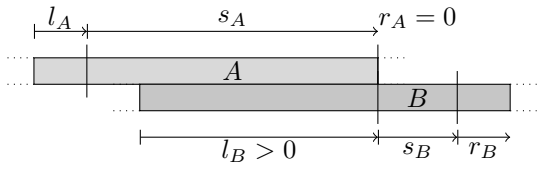

Figure 6: Internal synchronization constraints

that case, the value of s in the first encoding just equals the value of l_B in the second encoding. Even better, in that encoding, the left offset l_A of A (resp. the right offset r_B of B) remains unconstrained. It follows that both offsets can be used later for further internalized synchronization specifications of the resulting tiled signal $A; B$ with additional tiled signals, be them on the left or on the right.

In other words, with the increase of compositionality provided by the tiled signal approach, we still preserves the possibility of qualitative abstract reasoning on relative positioning.

2.5 The induced synchronization algebra

It occurs that the synchronization product $A; B$ defined above over tiled signals is an associative operation over tiled signals. The resulting algebraic structure is thus a semigroup. Aiming at defining interactive signal handling, with signals that are dynamically received, processed or synthesized, this is a much welcome property.

From a programing paradigm point of view, the synchronized product $A; B$ of two tiled signals A and B can be understood in two ways:

- at the abstract event-based layer : $A; B$ means that *"event" A is followed by "event" B*,
- at the concrete synchronous layer: $A; B$ means that *"signal" A is synchronized with "signal" B* with possible overlaps.

In other words, depending on the chosen time scale, every tiled signal can be seen both as an *asynchronous event* (on the logical time scale) or as a *synchronous signal* (on the synchronous realtime scale). In other words, the tiled signal approach is *multi-scale*.

The resulting algebra is described further in [5]. It is shown, in particular, that additional left and right RESYNC operators can be derived from the structure of tiled signals. They are depicted in Figure 7. Together with sequential

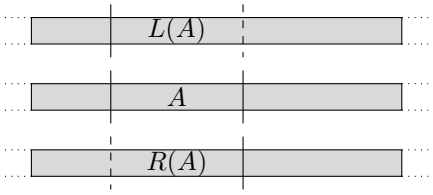

Figure 7: Left and right RESYNC operators

product, these resets operators considerably increase the expressive power of tiled signal expressions.

Indeed, one can define $FORK(A, B) = L(A); B$ with a synchronization of A and B at the beginning of their synchronization windows. On can also define $JOIN(A, B) = A; R(B)$ with a synchronization at the end of their synchronization windows. This situation is depicted in Figure 8.

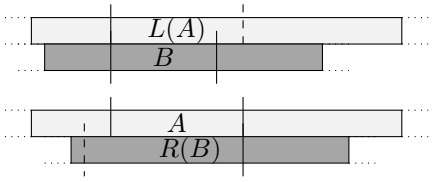

Figure 8: The FORK and JOIN derived operators

In other words, handling multi-channel signals that can seen both at the synchronous scale and at the event-based scale, our proposal thus provides descriptions of musical object in the three dimensional space $(T) \times (P) \times (A)$. The way the interaction dimension (I) is handled and experimented is the purpose of the remaining sections.

Remark. In the T-calculus presented in [12], the synchronization algebra is extended further with additional typed operators that can be applied to the synchronized product of tiles. Additional and rather subtle signal processing operators can then be derived.

3 Implementing the algebra

In this section, we describe the software components of the *libTuiles* library. In particular, we present the *libTuiles* API, the synchronous sound engine that is controlled by the asynchronous execution of the tuile, and an object-oriented architecture dedicated to messaging between musical threads.

3.1 LibTuiles: building and playing trees of tiles

LibTuiles is a C++ software library that allows for the creation and the execution of trees of synchronized tiles. In these trees, each tile is given an unsigned integer as unique identifier. The following methods of the class TuileManager are used to build and play the tiles.

addLeaf(const float& d, unsigned int& id) creates a new leaf tile with an initial length set to d and assigns its identifier to the id variable.

addLoop(const unsigned int& idChild, unsigned int& loopID) creates a new tile by applying the LOOP operator to the tile with the idChild identifier and assign the new identifier to the loopID variable.

addSeq(const unsigned int& idChild1, const unsigned int& idChild2, unsigned int& opID), **addFork**(...) and **addJoin**(...) create a tile by applying respectively the SEQ, FORK and JOIN operators to the tiles with the identifiers idChild1 and idChild2. The id of the resulting tile is assigned to the variable opID.

setTuileLength(const unsigned int& id, const float& l) applies the STRETCH operator with value l to the tile identified by id.

setTuileLeftOffset(const unsigned int& id, const float& lo) applies the RESYNC operator in order to set the left offset of the synchronization window of the tile with the id identifier.

setTuileRightOffset(const unsigned int& id,const float& ro) applies the RESYNC operator in order to set the right offset of the synchronization window of the tile with the id identifier.

setBpm(const float& bpm) sets the tempo at which the tree is played.

setRoot(const unsigned int& id) sets the tile with identifier id as root of the tree.

play() et stop() respectively starts and stops playing the tree.

removeTuile(const unsigned int& id) removes the tile with identifier id from the tree.

clear() removes all the tiles from the tree.

Internally, the manipulation and execution of the tree are done in a separate thread, in order to avoid slowing down when computations are done in the main application thread, for example within a graphical interface. The inter-threads communication mechanism is described in section 3.3.

When playing the tree, the temporal progression is computed in the root tile and spreads down the tree. Each operator computes the progression of its children based on the parameters of their synchronization and realization intervals. The play position in each tile is computed at any time t. Therefore, it is possible to know the absolute position of each tile within the tree. Because the temporal progression is computed for each node of the tree relatively to its parent node, it is also possible to dynamically modify the tree while playing it.

Activation and deactivation commands are sent from the playing thread respectively when tiles enter and leave their realization intervals. Lengths commands are also sent when STRETCH operators are applied or when the main tempo is modified. Absolute position commands are also sent whenever the tree is modified. Therefore, a synchronous audio synthesis/processing engine, such as the one described in section 3.2, receives all the commands required to temporally manage the processes associated to tiles.

Tiles properties can be accessed by calling the method **getTuileProps(const unsigned int& id)** which returns a structure associated to the tile with the identified id. This structure contains the various properties of the associated tile such as the length of the realization interval, the left and right offsets of the synchronization interval and the absolute position in the tree. This mechanisms allows for example for the update of tiles representations in a graphical user interface, as these properties may be impacted by manipulations of other tiles of the tree.

3.2 A synchronous engine for temporal structuring of musical processes

LibTuiles is connected to a synchronous synthesis/ processing engine based on the JACK sound server. This engine receives tiles activation/deactivation/length commands sent by the *libTuiles*. It then correspondingly activates/deactivates processes associated with the leaf tiles, these processes being nodes of an audio rendering graph.

Mainly two types of processes are handled by this engine. Sound file processes allow for the reading of sound files of any format handled by the libsndfile library. They also handle time stretching in order to match the changes in tempo and in tiles length without impacting the pitch of the sound, by relying on granular synthesis. At the initial speed, grains overlap by half and the position step between two grains is equal to half a grain. When the length of a tile increases, grain overlapping is increased and the step between grains is reduced and combined with a random offset in order to avoid artificial frequencies created by the proximity of grains. On the contrary, when the tile length decreases, the position step is increased together with the overlapping between grains in order to reduce amplitude variations between successive grains. This synthesis method, despite its quality being lower than other common time stretching methods, allows for both real-time stretching at a very low processing cost and also for click-free repositioning in sound files.

Leaf tiles may also be associated with FAUST processes. Connexions can then be made between processes or with the sound card inputs and outputs. Processing is only done when the input process and FAUST process temporally overlap, i.e. when the associated tiles are both active. Therefore the composition and properties of tiles allow for a fine temporal adjustment of the audio rendering graph.

3.3 Multi-scale object oriented system architecture

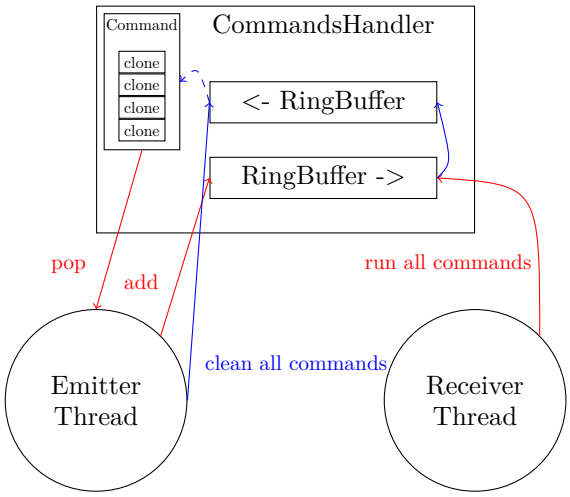

Figure 9: Software architecture for passing commands between two threads at different time scales.

One important aspect of the *libTuiles* architecture is the use of Commands, as depicted on Figure 9. These software modules allow for efficient communication between the event-based scale, the asynchronous real-time scale and the synchronous real-time scale, each of these scales being handled by a separate thread. In particular, the synchronous real-time thread that renders the audio signal does not tolerate interruptions that might be created by memory allocations and locking mechanisms. The proposed architecture relies on well-known object-oriented design patterns among which are the Prototype, the Abstract Factory and the Command. It also makes use of the ring buffer mechanism provided by the JACK library.

An instance of the *CommandsHandler* class handles the creation and manipulation of instances of classes that inherit from the *Command* class as well as their transmission from a sender thread to a receiver thread. This instance is therefore shared between the twothreads. Mappings between commands names and commands are first added to this class. For example the synchronous engine *CommandsHandler* includes commands such as *ActivateProcess* and *DeactivateProcess*. When a mapping is added, a prototype of the Command class is created. This prototype creates and holds a list of pointers to clones. In turn, each clone keeps a pointer to its prototype. For each message that needs to be passed from one thread to the other, a command can be simply defined by inheriting from the *Command* class and by redefining the *run()* method in order to manipulate data structures handled by the receiver thread, for example activating / deactivating processes.

During runtime, the emitter thread gets a pointer to a clone of a specific

Command by calling the *popCommand(commandName)* method of the *CommandsHandler*. The requested instance is then removed from the list of clones in the *Command* prototype and can be tweaked with various parameters, in our case the tile identifier, the new length of the tile and so on. As all clones are generated beforehand, no memory allocation is done in this call. The pointer to the clone is then given to the *CommandsHandler* and shared with the receiver thread using a ring buffer, in order to avoid locking mechanisms.

The receiver thread periodically calls the *runCommands()* method of the *CommandsHandler*. This method reads the *Commands* in the ring buffer, calls their *run()* method and send them back to the emitter thread through a second ring buffer. Finally, the emitter thread periodically calls the *cleanCommands* of the *CommandsHandler* which reads pointers from the second ring buffer and puts each clone back in the list of available clones of its prototype.

This object-oriented software architecture allows for passing commands between threads without memory allocation nor locking mechanisms. In addition, only pointers are passed through the ring buffer thus minimizing the memory consumption and transferring time. The *Commands* architecture is therefore particularly efficient for applications mixing different time scales, especially if these do not tolerate interruptions.

4 Interactive experiments

In this section we describe the interactive experiments conducted with the libTuiles. This is done via the *SimpleTuilesLooper* interface. Completing the text given below, a presentation video of this interface is also provided[1].

4.1 The *SimpleTuilesLooper* Interface with four leaf tiles among witch are three sound file tiles and one FAUST tile.

As depicted on Figure 10, *SimpleTuilesLooper* is an application that allows for the experimentation of temporal composition of sound processes, relying on *libTuiles* and on the synchronous engine described in the previous section.

This application sets a *Loop* tile as the root of a tiles tree, with a first leaf as child. All the other tiles added in the application are synchronized with this first leaf tile. Its synchronization interval, dynamically manipulable, defines the synchronization interval of the loop and therefore the looped interval when playing the tree. *SimpleTuilesLooper* allows one to create tiles associated with sound files and FAUST dsp files and to combine these tiles in order to build the tree using a *drag and drop* metaphor. These files are dragged from a file browser and dropped onto the score. Either they are placed freely on the score and internally composed using a fork operator with the root tile, or they are

[1] http://hitmuri.net/SimpleTuilesLooper

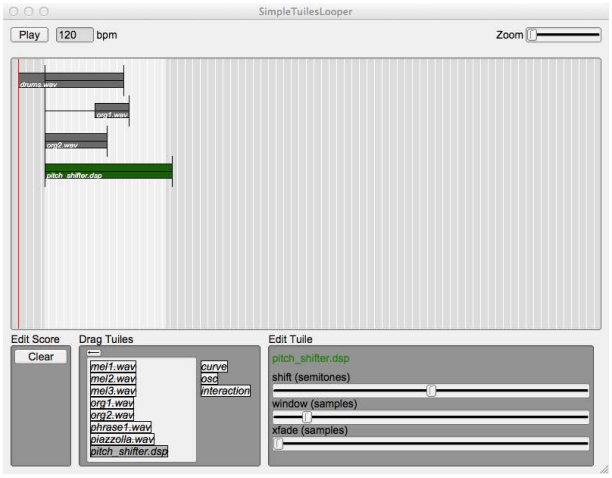

Figure 10: SimpleTuilesLooper allows for the temporal composition of sound processes by relying on libTuiles.

placed in fork, seq or join composition with an existing tile and properly inserted in the tiles tree.

The interface also allows for tweaking the FAUST effects parameters and for defining the connections between processes. The tree can then be played and dynamically modified by applying the RESYNC and STRETCH operators directly on the graphical tiles.

4.2 Monitoring tiled inputs and conditional tiles

Interactive dynamic tree manipulations are made possible by the use of monitoring tiles. These tiles are attached to listener processes that receive flows of audio samples or of MIDI or OpenSoundControl events and compares them with a number of predefined conditions. When one of these conditions is matched, a command can be sent to the *TuilesManager* to control either a monitor tile or a switch tile.

The monitor tile allows for dynamic sequential composition of tiles. It is similar to what can be done with trigger points in the i-score sequencer. When activated, this tile waits for a trigger event (or for the end of its realization interval). During that time, it does not play its child tile. When the event arrives, the monitor tile sets the length of its synchronization interval so that the end is at the current position, it then sequentially composes its child tile, and plays it when the child enters its realization window. The monitor tile therefore provides a way to adapt the progression in the composed tree to external events, for example coming from a musician or from the conductor.

The switch tile only plays one of its children, set by a method or command,

and uses the synchronization interval of the chosen child. Therefore, this tile allows for dynamic selection of a subtree among several subtrees, which is interesting for example in the case of structured improvisation with conditional branchings.

4.3 Loop tiles

Loop tiles are defined as infinite sequential compositions of children tiles with themselves. However, these tiles do not only repeatedly play their child, and therefore the associated subtree, within the synchronization interval. They also allow for interesting overlapping effects as described in [5], when a RESYNC operator is applied to the child of the loop tile. In the case of sound processes, this overlapping results in multiple instances of the audio result being played at the same time. It is therefore essential to provide a *polyphony* parameter for loop tiles. Interestingly, this parameter somehow provides a control over the resulting musical complexity. On the contrary to existing loop based formalisms such as the hierarchical live-looping [4] and to looping implementations in popular software instruments, here the looping mechanisms inherits from the properties of the composition operation defined within the tiles model, allowing for rich musical variations of simple patterns.

5 Conclusion

We described the implementation of an advanced synchronization algebra for audio or musical patterns. This software library, called the *libTuiles*, allows for the interactive creation, manipulation and execution of trees of tiled signals that embed a synchronization mechanism. Furthermore, it offers new musical possibilities, thanks to the underlying algebra, which can be experimented through a dedicated graphical interface. One of the perspectives of this work is to adapt the *libTuiles* so that it becomes the execution engine for the T-calculus [12] that extends the synchronization algebra.

References

[1] J. Allen and G. Ferguson. Actions and events in interval temporal logic. In Oliviero Stock, editor, *Spatial and Temporal Reasoning*, pages 205–245. Springer Netherlands, 1997.

[2] A. Allombert, M. Desainte-Catherine, and G. Assayag. Iscore: a system for writing interaction. In *Third International Conference on Digital Interactive Media in Entertainment and Arts (DIMEA 2008)*, pages 360–367. ACM, 2008.

[3] Antoine Allombert, Myriam Desainte-Catherine, and Mauricio Toro. Modeling temporal constrains for a system of interactive score. In G. Assayag

and C. Truchet, editors, *Constraint Programming in Music*, chapter 1, pages 1–23. Wiley, 2011.

[4] F. Berthaut, M. Desainte-Catherine, and M. Hachet. Drile : an immersive environment for hierarchical live-looping. In *Proceedings of New Interfaces for Musical Expression (NIME10)*, pages 192–197, Sydney, Australia, 2010.

[5] 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, december 2012.

[6] Alessandro Cipriani and Maurizio Giri. *Electronic Music and Sound Design - Theory and Practice with Max/Msp*. Contemponet, 2010.

[7] 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.

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

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

[10] D. Janin. Algebras, automata and logic for languages of labeled birooted trees. In *40th International Colloquium on Automata, Languages and Programming (ICALP)*, LNCS. Springer, 2013.

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

[12] D. Janin, F. Berthaut, M. DeSainte-Catherine, Y. Orlarey, and S. Salvati. The T-calculus : towards a structured programming of (musical) time and space. Technical Report RR-1466-13, LaBRI, Université de Bordeaux, 2013.

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

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

[15] P. Teehan, M. R. Greenstreet, and G. G. Lemieux. A survey and taxonomy of GALS design styles. *IEEE Design & Test of Computers*, 24(5):418–428, 2007.

[16] S. Wilson. *The SuperCollider Book*. Cambridge: The MIT Press, 2011.