Sound Processes: A New Computer Music Framework

Hanns Holger Rutz
IEM – Institute of Electronic Music and Acoustics
University of Music and Performing Arts
Graz, Austria
rutz@iem.at

ABSTRACT

Sound Processes is an open source computer music framework providing abstractions suitable for composing real-time sound synthesis processes. It posits a memory model that automatically persists object graphs in a database, preserving the evolution of these objects over time and making them available either for later analysis or for incorporation into the compositional process itself. We report on the experience of using a prototype of this framework for a generative sound installation; in a second iteration, a graphical front-end was created that focuses on tape music composition and introduces new abstractions. Using this more controlled setting allowed us to study the implications of using a live versioning system for composition. We encountered a number of challenges in this system and present suggestions to tackle them: the relationship between compositional time (versions) and performance time; the relationship between text and interface and between object dependencies and interface; the representation, organisation and querying of musical data; the preservation and evolution of compositions.

1. INTRODUCTION

The emergence of computer music systems is often tied to general developments in the computer science discipline, such as the establishment of new programming languages which serve as host languages or the appearance of new programming paradigms—e.g. object-oriented programming—that find their way into domain specific languages. Hardware developments also play a role, for example by making it possible in the mid 1990s to build new real-time sound synthesis systems for desktop computers. There is probably no abstraction or paradigm that has not been explored for its musical potential: Functional programming, dataflow programming, constraints and logic programming, concurrency abstractions, aspect-oriented programming, along with a number of design patterns.

On the other hand, the basic questions one has to answer when designing such a system appear to be unchanged. Already in 1976 Barry Truax listed the following: Which data representations are chosen, which operational capabilities, how is the flow of control organised and what are the input and output requirements? What are the structural levels and what is the granularity of access to the musical data, how can it be arranged and grouped? But also: What is the coverage of the system, to what extent does it intend to reflect the overall compositional process? [1]

1.1 Musical Representation

A great deal has been written about the representation of musical data, but some of the debate such as the age-old juxtaposition between procedural (implicit) and declarative (explicit) representation [2] has obscured more relevant aspects: The first concerns the understanding of representation as *knowledge* representation. Michael Hamman discusses this problem and defines 'representation' as something that «constitutes the agency through which an interface is embodied by orienting a particular way of conceiving and understanding a signal» [3]. If we rely solely on the established cultural denotation of representations, these might be *useful*, but we run into danger of confounding representation with the represented.

Second, taking the previous definition, it is clear that representations have a translational potency. A representation can always be rewritten as another, qualitatively distinct representation. For example, a procedural description of a sound production can be unfolded by following the procedure and recording its output, perhaps yielding an explicit sequence of events in time. Procedures in turn can be specified declaratively, giving rise to an abstraction such as the *dataflow variable*.

Finally, a representation specifies what is not represented. In the aforementioned article Truax made two important remarks: Before the advent of computer composition systems, the process of composing was difficult to assess, relying on artefacts such as the final score or at best sketch-book notes. The introduction of computer programs and the use of technical aids have resulted in «an increasing observability of musical activity», since these aids "externalise" the process. He then posits the thesis that any computer system embodies a model of the musical process; it becomes a "data source" for the study of musical activity.

The corollary that can be derived from these remarks is that a computer music system should take the *activity* of composing into account. But in the nearly forty years that have passed, the interest in musical representations has almost entirely focused on the way "musical time" is formulated—the time in which elements are placed during

the performance of a piece.

2. ACCOUNTING FOR CREATION TIME

Music software already stores data in a persistent way so that it becomes available for a later inspection. A number of experiments that observed composers at work asked them to store "snapshots" of these data, so that the evolution of the composition process could be examined. Apart from the coarse granularity of such sequences of snapshots, this approach requires an active intervention of the composer. As Christopher Burns notes:

> «Composers are generally more interested in producing work than in documenting it. Sketches and drafts are often saved only if their continuing availability is necessary for the completion of a project, and mistakes and false starts are unlikely to be preserved.» [4]

My main critique however concerns the usage of the data thus obtained. Truax reserves the observation to "theorists" who seek to understand the musical activity, whereas the composers themselves are not mentioned. The externalisation of the storage action means that the historic trace of the decision-making process itself has no useful representation within the composition system itself. There is no re-entry of the temporal embedding of the decisions within the decision-making process. This is also implicit in Burns' reflection that assumes a complementarity between production and documentation.

As an analogy, we can look at the process of software development. Today it is not possible to imagine this process without the employment of version control systems such as *Git* or *Subversion*. These technologies have multiple goals, including the review of decisions in order to find mistakes and the possibility for multiple users to concurrently manipulate the code base and eventually "merge" their work. What is not provided is for the developed software to engage with its own history, so there is no interface back from the versioning system to the developed software.

This is probably fine, since versioning is just a "tool" in the software design process that helps to achieve the design goals. In computer composition, however, questions of representation—the data structures, their interfaces and relations—are the very *materials* of the composition itself. Hamman, in looking at Agostino Di Scipio's work and that of Gottfried Michael Koenig, argues that «just as one might compose musical and acoustical materials *per se*, one might also compose aspects of the very task environment in which those materials are composed.» If the process of decision-making is itself made manifest within the composition system, it can re-enter that process as one of its possible materials.

To distinguish the different temporal ascriptions of a datum, we proposed the following terminology: [5]

- The (actual) performance time $\mathcal{T}_\mathsf{P}$. When a musical datum is heard in a "real-time" performance, this happens in $\mathcal{T}_\mathsf{P}$.
- The virtual performance time $\mathcal{T}_{(\mathsf{P})}$. This is the representational form of $\mathcal{T}_\mathsf{P}$. For example, if we think of a timeline view, the positions of elements on the timeline are values in $\mathcal{T}_{(\mathsf{P})}$.
- The creation time $\mathcal{T}_\mathsf{K}$. This is the time when an object is created, modified or deleted as part of the composition process.

In *Sound Processes* the primary concern is the handling of $\mathcal{T}_\mathsf{K}$ as it informs the underlying memory model. The data structures employed and their interaction have been described before [6], thus we just give a brief overview.

2.1 A Memory Model for Sound Processes

The memory model is an extension of software transactional memory (STM). In STM, the basic unit of operation is a reference cell that stores a value. The two permitted operations are access (reading the value) and update (writing or overwriting the value). This value can be either an immutable entity such as a number or a pointer to another reference cell. The operations must be performed within a *transaction* that provides the properties of atomicity, consistency and isolation: Multiple operations performed inside the same transaction form one compound and indivisable operation. If an error occurs, all operations participating in the transaction are undone together.

Transactions are also used in databases, and since version control systems utilise databases, there are similarities between an STM and a VCS. Similar to the snapshot scenario above, in a VCS the user explicitly decides when to make a new snapshot. This action is called *commit*. This is a manual transaction and it is the responsibility of the user to maintain some sort of "consistency" for the state of the code base at the moment of committing. Each commit is tagged with a user identifier and a time stamp and constitutes a new *version*. The VCS allows one to create new branches from any previous version and to merge multiple branches into one, producing a version graph.

In *Sound Processes*, the STM is extended with the semantics of a versioning system: Each transaction is associated with a time stamp representing $\mathcal{T}_\mathsf{K}$, and the evolution of the reference cells is automatically persisted to secondary memory (hard-disk). From the user's perspective, these cells still look like ordinary STM cells, but they have to be accessed through special transaction handles provided by so-called *cursors*. A cursor represents a path into the version graph, and when a cell is accessed or updated, behind the scenes a complex index resolves the history of that cell to find the value associated with it at the particular moment in $\mathcal{T}_\mathsf{K}$. From the system's point of view, it makes no difference whether one looks at the most recent "version" of a composition or any other moment in its history. Moreover, we can now programmatically ask when a datum was modified or what its past states were, and we may use this information in an artistically meaningful way.

3. SECONDARY DATA STRUCTURES

On top of this fundamental level of an automatic and concomitant versioning, arbitrary structures can now be defined for the "intrinsic" musical data.

Many authors have taken up on the distinction between *intime* and *outside-time* data prominently expressed by Iannis Xenakis. We can now say that "outside-time" only refers to $\mathcal{T}_{(\mathsf{P})}$. The composer conceptually "spatialises" material, i.e. organises it in some form of tableau or collection of things which can be manipulated prior to assigning them positions in $\mathcal{T}_{(\mathsf{P})}$.

3.1 Expressions

We provide simple data types for numbers, boolean values, strings etc. along with tuples and ordered and unordered collections. In order to be able to establish relationships between such elements, we create a dataflow-like layer. Here objects can propagate changes to their dependents. Unlike variables in common dataflow programming languages whose values are initially unknown and will be assigned only once, we use the concept of expressions that have an initial value and may be updated multiple times. Thus they closer resemble objects in a *PD* or *Max* patch.

Without loss of generality, we propose to represent points in $\mathcal{T}_{(\mathsf{P})}$ as expressions whose value is of type `Long`, a 64-bit integer number representing an offset in sample frames at a chosen sample rate and logical offset. Time intervals use type `Span` which can be thought of as a tuple of a start and a stop point in time. Unbounded intervals are also permitted, e.g. if an object is created in a real-time live situation, it may have a defined start point but an undefined end point. If the object is eventually deleted, the span is updated with a defined end point.

The following code is an example of how a programmatic creation of an expression tree looks like. It defines a function that ties a span `succ` to an arithmetic expression formed by an offset `gap` appended to another span `pred`:

```
def placeAfter(pred: Expr.Var[S, Span],
               succ: Expr.Var[S, Span],
               gap : Expr    [S, Long])
              (implicit tx: S#Tx): Unit = {
  val newStart = pred.stop + gap
  val newStop  = newStart + succ().length
  succ()       = Span(newStart, newStop)
}
```

A visualisation of the structure is shown in Fig. 1. In short, an object `Expr[S, Long]` is an expression in system S which evaluates to a long integer. Different systems can be used to decide whether a structure should be traced in $\mathcal{T}_{\mathsf{K}}$ or not. An `Expr.Var` is a variable holding an expression. The broken arrow results from reading the old value of `succ()` for determining the length of the updated span. The graph is thus acyclic—cyclic object graphs are currently not supported.

3.2 Sounding Objects

The symbolic nature of programming languages naturally produces a bias towards supporting symbolically represented structures. To improve on the support for electronic and electro-acoustic materials, we base our core abstraction for sounding objects, `Proc`, on three members:

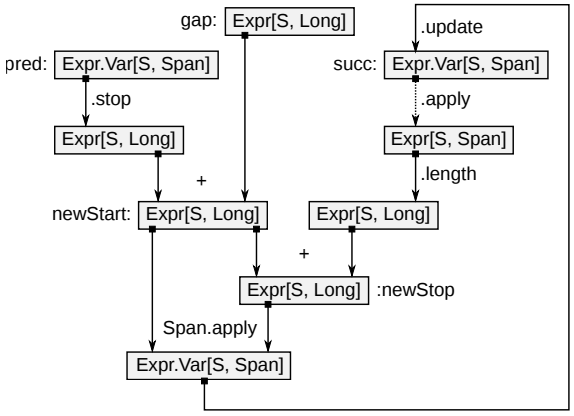

Figure 1. Expression chains produced by function `placeAfter`. Arrows point in dataflow direction from dependency to dependent.

1. An expression `graph` that evaluates to a unit generator graph handled by the *ScalaCollider* library, a client for the *SuperCollider Server*.
2. A dictionary `scans` that maps between logical signal names and real-time input or output signals.
3. A dictionary `attributes` that maps between logical key names and heterogeneous values used to configure the sound process.

The unit generators are extended by various elements which interact with the `Proc` structure, for example by reading from a scan input, writing to a scan output, determining the placement of the process in time, accessing the `attributes` dictionary, etc.

A 'scan' is a connecting point, it administrates sinks (process outputs) and sources (process inputs). A sink or source may be either a grapheme or another scan. A grapheme is a random access object—accessible both in real-time and offline—producing a linear time signal from segments of break-point functions or stored audio files. A scan signal is produced either by linking the scan's source to another scan's sink—thus establishing "bus routing" between processes—or a grapheme input, or it is produced by the process' graph function itself. This is illustrated in Fig. 2. [1]

Processes are placed in $\mathcal{T}_{(\mathsf{P})}$ by associating them with a time span—which may be an expression and thus algorithmically specified and updated. A special data structure keeps a designated group of processes indexed in $\mathcal{T}_{(\mathsf{P})}$, and a transport class may then iterate over this temporal dimension in real-time (or offline for the purpose of bouncing).

4. VOICE TRAP

Several pieces were realised using the system. We report on two of them: A sound installation *Voice Trap*, written

[1] The dashed arrow from grapheme to graph means that the implementation for plugging graphemes directly into sinks is currently missing, but that there are work-arounds to record the real-time signal and introduce the recording as a new grapheme.

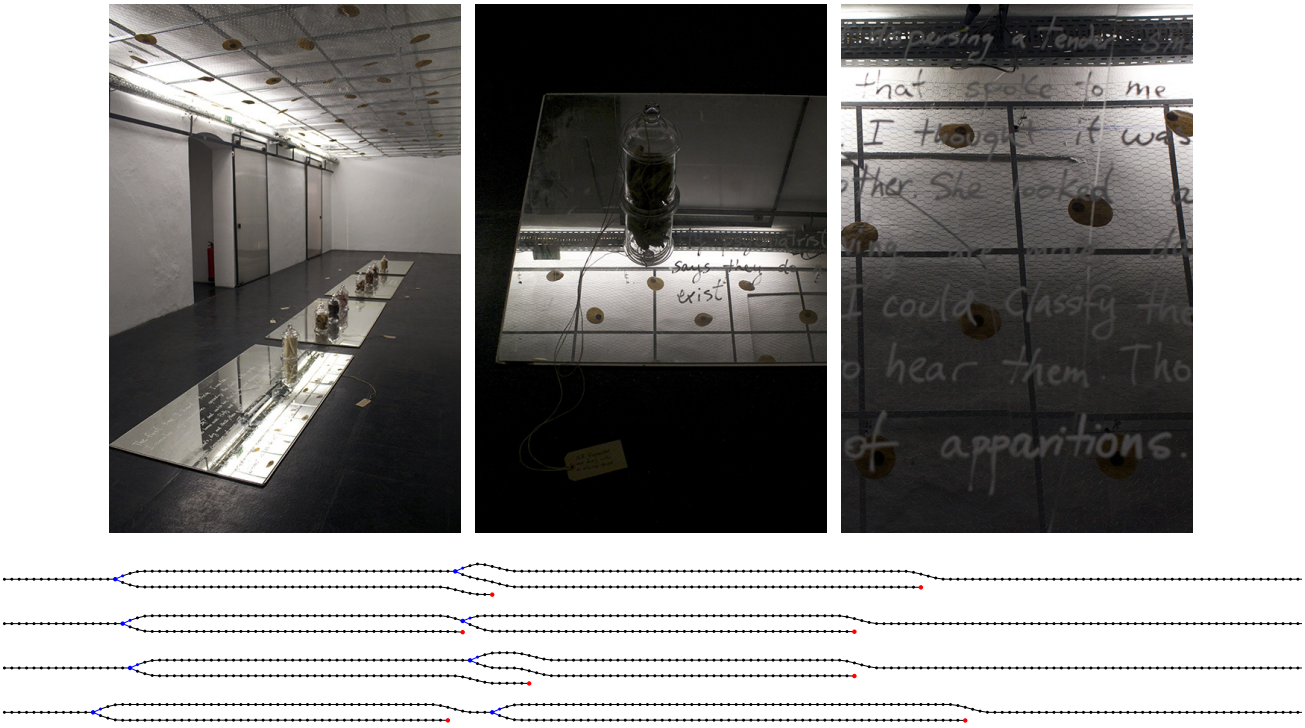

Figure 3. Wide shot and details of *Voice Trap* (top), and version graph detail (bottom)

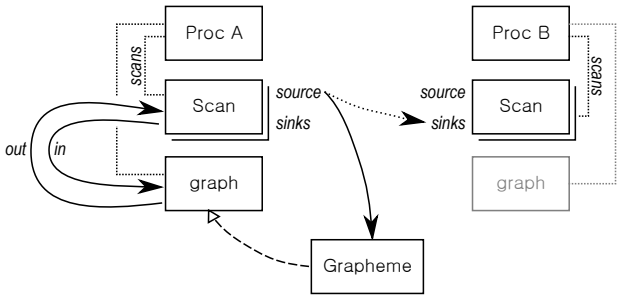

Figure 2. Interaction between scans, graphemes and graph functions

using the "bare-bones" framework, and a tape composition *(Inde)terminus*, written using an emerging environment with a graphical front-end.

Voice Trap is a collaboration between me and visual artist Nayarí Castillo. It is spun around the story of a girl who is haunted by voices. The story is written across four large mirrors on the floor of the room. Large jars, "voice traps", are filled with different materials and placed on the mirrors. The jars are tagged with the written description of a particular voice and their contents relate to the sound qualities of the imaginary voices. The sound installation is diffused from 96 piezo speakers grouped into twelve channels which are placed on a metal grid suspended below the ceiling. Fig. 3 shows photos of the exhibition.

The material of the sound composition comes from a microphone that picks up the noises from the street in front of the gallery. These are fed into a database from which individual phrases are constructed. An algorithm searches the database for sounds that are both similar to the currently playing sounds as well as to an inaudible "hidden" file containing different voice recordings. The idea is that from the outside sounds those fragments will be preferred which contain speech. Each of the twelve channels operates independently; the evolution of each channel is captured by our framework, and the algorithm can make references to this history.

The bottom of Fig. 3 shows an example version graph for four channels. Each channel has a dedicated cursor, and each horizontal stretch is the succession of transactions producing a certain number of iterations over the sound phrases followed by a jump into the "past", going halfway back between the current transaction and the last branching point. After a jump back in $\mathcal{T}_K$, the sound phrase from that past version is heard again, but the successive evolution (overwriting of fragments with new sounds) diverges from the previous path, because the sound database itself is ephemeral and not reverted to a previous state.

Although I found it difficult to perceive these jumps—perhaps due to the channel-locality of the jump or due to the fact that the specific environmental sounds are more difficult to distinguish than traditional musical gestures made from pitches—this piece demonstrated that the framework is functional and can handle a continuously growing database even after tens of thousands of transactions and several hundred megabytes file size.

There was no specific development environment that allowed the composition of the algorithms in a traceable way; they were written in the object language using a traditional IDE, an activity which remained unobserved. On the other hand, the traces the algorithm produced inside the observed domain were easily captured. Constructing a whole meta language was too much of an effort at this stage, so another

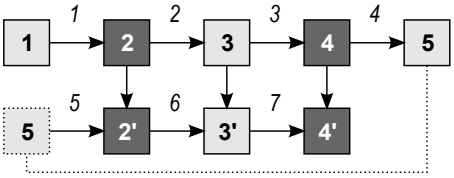

Figure 5. Procedure of *(Inde)teminus*. The edge labels indicate the sequence in $\mathcal{T}_K$. The fifth iteration replaces the first iteration in the first recursion—second row—which "rewrites" iterations 2 to 4.

path had to be taken to validate the approach in a more constrained setting.

5. (INDE)TERMINUS

Such a setting was established in another experiment. Its working title *(Inde)terminus* refers to Gottfried Michael Koenig's tape piece *Terminus I* from 1961 which is based on a scheme for deriving sounds from previous sounds by applying a set of transformations [7].

To realise this electroacoustic study, a graphical tape music environment named *Mellite* was written, based on *Sound Processes*. A screenshot is shown in Fig. 4. On the left side, a timeline view can be seen with several audio file regions placed on the canvas. The supported operations are: adding and removing, selecting, moving, resizing, muting or un-muting a region, adjusting its gain and fade curves.

We use the concept of a workspace which is a tree of "elements", shown as a window on the right-hand side of the screenshot. The opened popup menu shows the types of elements supported: folders, process groups (timelines), artefact stores (hard-disk locations), audio files, text strings, integer and decimal numbers, and code fragments. Elements can be dragged and dropped between different locations of the interface.

The code fragment elements played an essential part. The experiment begins with an initial hand-constructed canvas of three minutes duration, sparsely placing sounds on an 8-channel layout. In the next step a bounce is carried out and fed through a signal processing stage, becoming the blueprint for the next iteration. Here, a new canvas is built around this blueprint, possibly cutting it up, removing some parts of it and adding new sounds. Then again a bounce and a transformation is carried out, and so forth. This is illustrated in the top part of Fig. 5.

The environment uses an embedded *Scala* interpreter and an integrated code editor to textually manipulate objects or, in this case, to define transformations of the bounced sounds. The *creation procedure* of this transformed sound file is memorised, so it can be re-rendered at a later point even if the input canvas has changed. This idea is illustrated in Fig. 6 and works as a generalisation of the expression cells, whereby the deployed sound file artifact serves as the "evaluated" expression.

In this study the transformation was a segmentation and reversal of the resultant segments of the bounced file. Each channel was bounced and transformed separately, leading

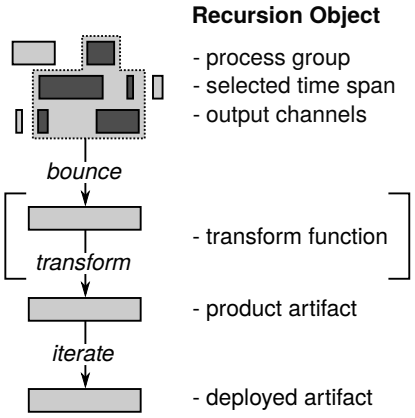

Figure 6. Algorithm for the transition to the next iteration

to different segmentations so that not only a diachronous reversal occurs, but also a synchronous scattering. The transformed bounces were placed on a new timeline and cut again into chunks to remove the silent parts. The new temporal structure was then adjusted and "composed", possibly thinning out the material further or introducing new elements. Since the next iteration would again reverse the temporal succession, a specific similarity arises within the group of even-numbered iterations and within the group of odd-numbered iterations.

The trace of the re-imported bounces permitted the creation of a closed recursive setting: After a certain number of iterations, the input to the initial bounce is exchanged for the result of the most recent (fifth) iteration, *retroactively* re-triggering the bounce and transformation of Fig. 6. Consecutively, the iterations would be re-worked, a procedure that could be repeated ad infinitum, explaining the title of the study. Practically, this re-working was carried out for the second (sixth), third (seventh) and fourth (eighth) iteration, as shown in the bottom row of Fig. 5.

The "flattening operation" of the bounce establishes what may be perceived as a *crucial deferral* or *suspension* in the process: A time canvas is manipulated whose product is used in another canvas, but the propagation of the changes from the former to the latter is suspended until a conscious decision is made. Furthermore, the flattening bounce provides the closure of the material which makes it possible to subject it again to general transformations such as the segmentation and recombination. This connectivity is an important feature of a representation, perhaps more important than its "symbol" function (Hamman).

5.1 Ex Post Analysis

From an outside perspective, the version history can now be used to query different aspects of the process. As an example, Fig. 7 shows a "punch card" plot similar to the ones given by popular open source platform GitHub. It indicates at what times of the week someone has worked on a piece of software. While composing is hardly an office job, charts like this, especially when more data is available, could reveal different profiles of composers, or they could be used to compare different types of activities.

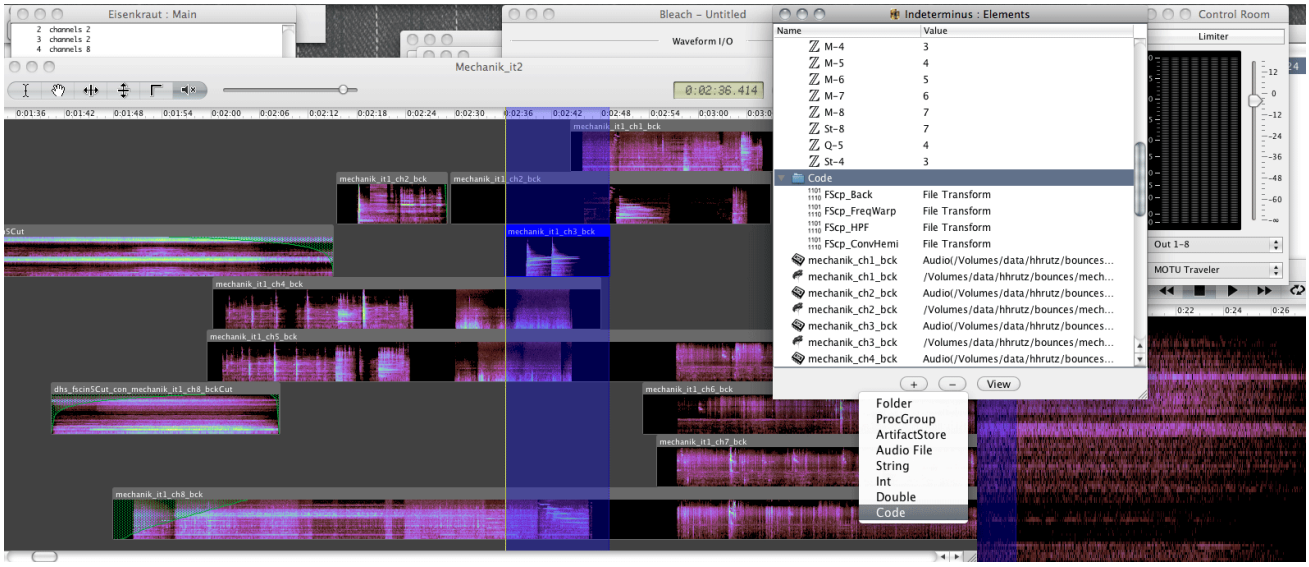

Figure 4. Screenshot of the *(Inde)terminus* session

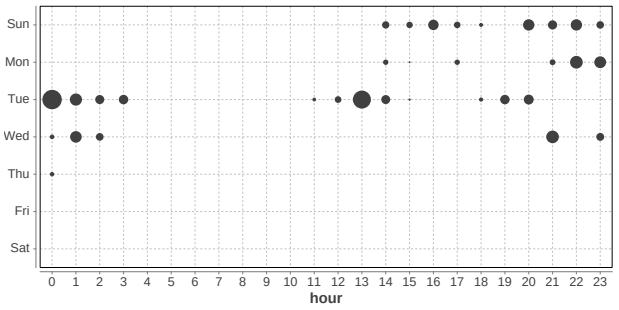

Figure 7. "Punch card" of working hours distribution

While the selection of sound files was not important to the concept of *(Inde)terminus*, we used a query in a different composition to reveal when particular sound files had been added to the piece, and it was possible to further elucidate this trajectory in $\mathcal{T}_K$ by taking manual sketch-book notes into account.

Another approach is to look at the tool usage. Fig. 8 shows the relative proportions. Extracting this data from the database was laborious, because the transactions were not specifically tagged by the software and needed to be reconstructed by analysing the structural differences between successive points in $\mathcal{T}_K$. To see how the proportions change over time, two charts were generated. The top chart shows the earlier transactions, relating to the first two iterations of the experiment. The bottom chart relates to iterations 2–4 after the recursion (or the second row in Fig. 5).

Moving and resizing amounts to more than half of the actions performed.[2] As the process unfolds, the relative number of additions, removals and especially movements decreases. This is in accordance with the idea to let the bounce transformation create a temporally reversed structure by itself and to accept that structure as a basis of each new iteration. In contrast, the number of region splittings,

[2] However, the extraction of data for gain change actions was difficult in this particular study and is omitted in the charts.

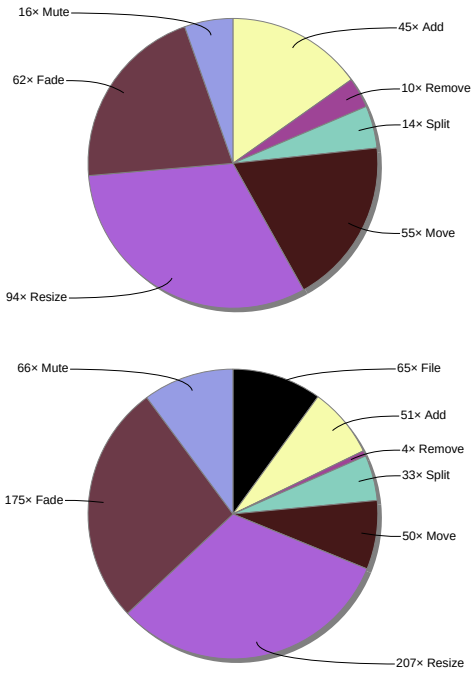

Figure 8. Frequencies of *Mellite* tool actions in the beginning (top) and at the end of the study (bottom)

adjustments of fade curves and mute/unmute actions goes up. It may indicate more work on the detail of the sound as well as an increased density of sounds that requires to mute sounds temporarily in order to monitor these details. Muting can also be used as an alternative to removing regions. The black pie segment labelled "File" indicates the actions of replacing the previously deployed artefacts with the updated artefacts.

One can also look at the parametrisation within the groups of actions. Fig. 9 shows the distribution of varieties among the resize actions and the region movements. The histogram bins use a logarithmic time scale and labels give

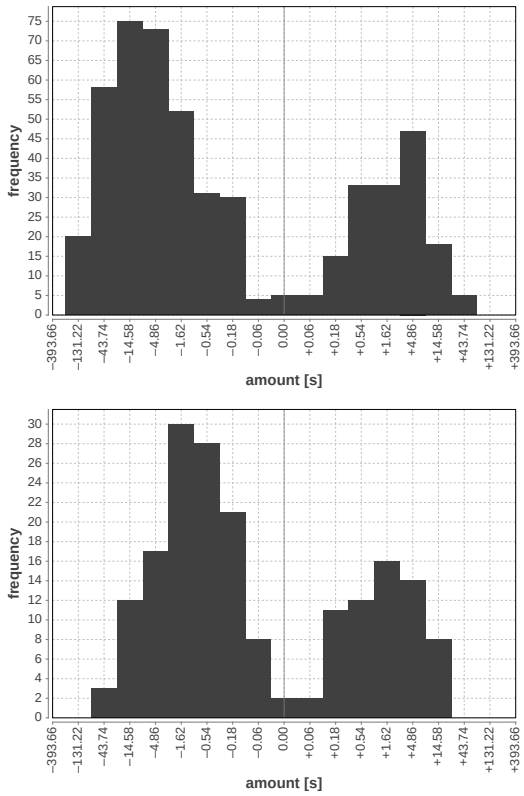

Figure 9. Distribution of the amount of contraction and expansion in resize actions (top) and relative time shift in move actions (bottom)

the lower interval margin. The vertical line in the centre distinguishes contractions on the left and expansions on the right (resize) or shifting backward in $\mathcal{T}_{(\mathsf{P})}$ on the left and forward in $\mathcal{T}_{(\mathsf{P})}$ on the right (move). An interesting "left-leaning" tendency can be observed in both cases: Regions tend to be shortened rather than elongated, but also material moves backward in time more often than forward, perhaps due to an editing style which initially gives each region some isolated space before condensing the structure left-to-right. Besides, there is an overall bell shape in the distribution of both action types, which may be inherent to the type of sound material used or dominated by the typical zoom levels used in the graphical interface.

Another analysis examined the development of the statistical moments of the regions' durations over time, and some characteristic motions and settlements could be observed. These charts and their discussion have been omitted for reasons of space. There are many more possible ways of extracting information from the database: One could compare composer with composer, piece with piece, sections within a piece, sections within the creational timeline; one might use such information to test or support hypotheses about the working process, the musical material or the human-computer interaction. The beauty of this approach lies in the fact that the situation is not a priori contaminated with "musical meaning" or "musical interrogation", but indeed accentuates motions which underlie the compositional process and which may otherwise remain tacit.

Finally, we created a transcription that brings together $\mathcal{T}_{\mathsf{K}}$ and $\mathcal{T}_{(\mathsf{P})}$. Such "motiongrams" are shown in Fig. 10. The blackening corresponds horizontally with the span in $\mathcal{T}_{(\mathsf{P})}$ affected by an action at a given vertical point in $\mathcal{T}_{\mathsf{K}}$. The different iterations of the experiment are preserved as horizontal segmentation. One could interpret that diagram again. One would find the "carriage returns" in scanning through the timelines; discern the initial phase of each iteration from the subsequent refinement; see moments of obstinate distillation at a particular spot; see at which point in $\mathcal{T}_{\mathsf{K}}$ a certain part of the piece is more or less finished…

6. LIMITATIONS AND FUTURE DIRECTIONS

Drawing from the experience gathered so far, we will now highlight some limitations and make suggestions for future refinements of the framework. First of all, the querying possibilities should be improved and extended, especially for collections: Finding out when elements were added or removed requires iteration over the whole data structure for each possible version step. What we envision is a general indexing operation that produces auxiliary data structures for ordered or unordered sequences. One should be able to index a group of sound processes not just by their positioning in $\mathcal{T}_{(\mathsf{P})}$ but by arbitrary parameters such as creation or modification date in $\mathcal{T}_{\mathsf{K}}$, timbre or dictionary key. Collections should also allow the application of (dynamic) filters.

Indices must be kept up-to-date and in- or revalidated when a key changes. An infrastructure for forward dependencies between objects already exists due to the event bus system that drives the dataflow and expression types. The more experience we gain from using the system, the more desirable it seems to extend the memory model with an automatic way to trace forward references. For example, if an object is "deleted", we might want to determine any other locations within the workspace that refer to this object. Does the composer wish to remove the object only in one particular place or across the workspace? A forward reference mechanism more general than event passing yields a form of automatic garbage collection. We imagine that the next iteration of the framework will implement a simple form of GC such as reference counting.

Another consequence of forward references is that the serialisation mechanism must be adapted. It is currently a statically typed and strict top-down approach. While it has many advantages, it cannot handle "blind" bottom-up deserialisation which would be needed for these forward references, and it is hard to extend the expression system in an open-ended way. Blind deserialisation would also ease the exporting of data to other formats and the automatic deep traversal of data structures, something that would allow the copying of objects from one workspace to another. Currently, workspaces are isolated from each other.

In terms of programming paradigms, a generalisation of the dataflow model with logical variables modelling constraint satisfaction problems (CSP) seems an interesting direction. These types of variables are initially only known by their bounds or the domain of values they can *possibly* take on. Comparable to the way in which we construct expression chains with single valued variables, these logical variables can be composed, and special operators establish

Property	Current state	Proposal
Memory disposal	Manual	Garbage collected
Serialisation	Static, top-down	+ Dynamic
Cyclic graphs	No	Yes
Indices	Specific	Generic
Timeline objects	Non-nested	Nested, recursive
Expressions	Determinate	+ Constrained
Workspace	Isolated	Interacting
User	Single	Collaborative

Table 1. Suggestions for improving the framework

constraints between them. Instead of saying that a sound object starts *this much* time after another sound object (the `placeAfter` example), we can just generally say that it starts *after* that sound, or we could say it starts *at most* this and this much time after that sound.

In terms of the representation of musical data, we feel that the current timeline model is too limited. A more powerful representation would allow the hierarchic and recursive nesting of elements in $\mathcal{T}_{(\mathsf{P})}$. Similar to the idea of filtering collections as an expression operator, fragments of one timeline could appear within an outer timeline.

In terms of usage scenarios, the studies have shown that the framework scales reasonably well to be used for real-time generative sound installations as well as mixed offline/online work such as tape composition. We have also developed a real-time graphical user interface for live improvisation, but it has not yet been coupled with the current version of *Sound Processes*, a case we still have to explore.

A second scenario is the collaboration of multiple composers on a composition, or performers improvising together; can we associate transactions with different users? What is the nature of distributed transactions or do we need to constantly merge multiple distributed transactions?

The previous suggestions have been summarised in Table 1. Of course, there are many more paths to explore. Graphical user interfaces is one of them. How should interconnected dataflow expressions be represented and edited? How do we convey links and dependencies between different elements across the user interface, without resorting to "patch cords"? How continuous are the transitions between a live improvisation view and a tape editing view? What is the relation between code fragments and graphical, symbolic or iconic elements?

7. CONCLUSIONS

We concluded our previous paper [6] by saying that the most important task would be to put the framework into production in different contexts and see how it scaled under real-world conditions. We believe this task has been successfully completed, and the current paper showed that a great number of interesting questions arise from the possibility to concomitantly trace the version history or to analyse it ex post facto.

Our next research focuses on the challenges and suggestions described in the previous section, as well as the extension of the *Mellite* front-end to a full-blown environment usable by other composers. The conflict between such *usability* and the critical value software plays in the artistic episteme is aptly worded by Hamman: [3]

> «When well-designed, the interface should tell us, by reminding us of our history of experience, how it works. We shouldn't have to think about how to use a door knob, for instance … At precisely the moment when an interface becomes sensible and useful, however, the shapes, materials, and structures which constitute its physical and epistemological frame, cease to exist in themselves…»

We should thus not forget the advantage of having—and retaining—a prototypical situation that can be understood as a "foregrounding" of representations, viewing music composition «as a task that is as much concerned with the theories and procedures by which musical artifacts might be generated as it is with the actual generation of those artifacts.» (Hamman)

Acknowledgments

The research was supported by a PhD grant from the University of Plymouth. The *(Inde)terminus* study was carried out during a studio residency provided by ZKM Karlsruhe.

8. REFERENCES

[1] B. Truax, "A communicational approach to computer sound programs," *Journal of Music Theory*, vol. 20, no. 2, pp. 227–300, 1976.

[2] T. Winograd, "Frame representations and the declarative/procedural controversy," in *Representation and Understanding: Studies in Cognitive Science*, D. G. Bobrow and A. Collins, Eds. New York: Academic Press, 1975, pp. 185–210.

[3] M. Hamman, "From Symbol to Semiotic: Representation, Signification, and the Composition of Music Interaction," *Journal of New Music Research*, vol. 28, no. 2, pp. 90–104, 1999.

[4] C. Burns, "Tracing Compositional Process: Software synthesis code as documentary evidence," in *Proceedings of the 28th International Computer Music Conference (ICMC)*, Göteborg, 2002, pp. 568–571.

[5] H. H. Rutz, E. Miranda, and G. Eckel, "On the Traceability of the Compositional Process," in *Proceedings of the 7th Sound an Music Computing Conference (SMC)*, Barcelona, 2010, pp. 38:1–38:7.

[6] H. H. Rutz, "A Reactive, Confluently Persistent Framework for the Design of Computer Music Systems," in *Proceedings of the 9th Sound and Music Computing Conference (SMC)*, Copenhagen, 2012, pp. 121–129.

[7] G. M. Koenig, "Genesis der Form unter technischen Bedingungen," in *Ästhetische Praxis*, ser. Texte zur Musik. Saarbrücken: PFAU Verlag, 1993, vol. 3, pp. 277–288.

Figure 10. Motiongrams for *(Inde)terminus*. The iterations are shown from left to right, transactions advancing from top to bottom. In each diagram, the horizontal extent covers the canvas duration of the particular iteration. Dotted lines indicate the beginning of the recursive re-workings. If an invisible grid is superimposed, the matrix of Fig. 5 can be seen.