A REACTIVE, CONFLUENTLY PERSISTENT FRAMEWORK FOR THE DESIGN OF COMPUTER MUSIC SYSTEMS

Hanns Holger Rutz
Interdisciplinary Centre for Computer Music Research (ICCMR)
hanns.rutz@plymouth.ac.uk

ABSTRACT

The process of composition can be seen as sequence of manipulations on the material. In algorithmic composition, such sequences are prescribed through another set of sequences which yield the algorithm. In a realtime situation the sequences may be closely linked to the temporal sequence of the unfolding musical structure, but in general they form orthogonal temporal graphs on their own. We present a framework which can be used to model these temporal graphs. The framework is composed of layers, which—from low to high level—provide (1) database storage and software transactional memory with selectable temporal semantics, (2) the most prominent semantics being confluent persistence, in which the temporal traces are registered and can be combined, yielding a sort of structural feedback or recursion, and finally (3) an event and expression propagation system, which, when combined with confluent persistence, provides a hook to update dependent object graphs even when they were constructed in the future. This paper presents the implementation of this framework, and outlines how it can be combined with a realtime sound synthesis system.

1. INTRODUCTION

When using the computer as a composition system, the choice of data structures for the representation of musical objects and processes (the interaction between objects) delimits what can be expressed. A system may be designed with symbolic processing or realtime sound synthesis in mind, or it may have structures to represent portions of sound files as in a tape editor. Several dimensions may be nominated, for instance the degree of expressivity in the representation of temporal values, or the overall paradigm such as declarative, procedural, functional, object-oriented (cf. [1]). What remains implicit in such a classification is that data structures determine not only which relations are represented, but also the *mechanism* by which musical objects are instantiated and manipulated, and how the *process of composition* itself is modelled.

We present a configuration of data structures which particularly focus on this second delimitation. These structures are designed to preserve the temporal trajectories formed by the process of composition. They store information about the temporal succession in which musical structures are created, where creation refers both to "editing actions" carried out by the composer in some software environment, but also to compositional algorithms which re-evaluate expressions (for example a random number generator) or transform the structure of the composition. While a lot of research has been conducted to establish models of representation of musical time, none of these studies take into account the temporal trajectory formed by the activity of composing itself.

This paper aims to rectify this deficiency, opening up new possibilities both for composers and researchers to observe this activity. Most importantly, beyond the preservation of the history of the creation of a piece—which perhaps never terminates, as in a realtime generative installation—, structural feedback processes are enabled, where recursion may use fragments from any previous point in the version graph of the piece. A possible scenario is a self-organising work which may then «interact with its own history» [2].

In section 2, we give an overview of the concepts and technological foundations of the proposed framework. In section 3, the actual implementation and the interplay of components is discussed. Finally, section 4 introduces a work in progress, the coupling of this framework with a realtime sound synthesis system. While this system has not yet been used in production and therefore no experience report or benchmarks can be provided, it builds on the design of a previous ephemeral (memory-less) version, which has been used in live-electronic pieces and sound installations.

We subscribe to the notion that «Knowing how to program is the key to doing something really new in computer music» [3, p. 3], implying that a composer using computer music programming languages will benefit from engaging with these concepts to make them productive in the field of computer music. However, the reader should not feel deterred if certain parts of the technological discourse remain foreign, as he or she should still be able to take away to contours of this new approach to modelling the process of composition.

2. CONCEPTS

2.1 Two Layers of Time

When investigating the relationship between the time attributed to musical data as part of a score or performance, and the time within which the composer actually writes

that "score", a useful analogy can be drawn to *bi-temporal databases*, and we can use the conceptual framework of the latter: In those systems, the actions of entering, modifying or deleting data form a timeline which is called *transactional*, while the data itself can be tagged with temporal values which define a *valid* time—the time at which the data has meaning in "reality" [4]. For instance, if the data set describes a person, a valid time span may be attached to that data set, specifying from when and until when the person was employed by a company. Transactions are typically considered atomic in that they represent decisions which do not occupy logical time themselves, while values in the valid time domain—as seen in the previous example—are often durational spans.

In a tape music editor, actions such as importing a sound file, placing it on the timeline, cutting and splicing it, have atomic character. The shape and velocity of the movement of the mouse cursor is not registered. What is registered is that a region *was* at a specific position in the timeline before the drag, and that *after* the drag it it occupies a different position. The abstraction from the actual movement of the region is not only useful to minimise the information which must be stored, but it also ensures consistency. A drag gesture can be aborted, and everything returns to its previous state, so there is no ambiguity with respect to the region's position.

Furthermore, the composer may work in a random access fashion, therefore the succession in which the timeline is filled in no way needs to reflect the order of the elements *on* the timeline—which represents the *prospective* valid (to-be-performed) time.

Atomicity and consistency are two of the four properties—known by their acronym ACID—which define most database systems, the remaining two being isolation and durability [5]. Isolation means that the system can process concurrent transactions safely so that each transaction sees a consistent view of the dataset isolated from any ongoing modifications by another transaction. Concurrency may be an issue in an system where multiple algorithms operate in parallel. Durability ensures that the effects of a transaction, after it completed, are permanently saved (e.g. on the hard-disk).

Three of these four properties, ACI, are also obeyed by Software Transactional Memories (STM). An STM can help to design systems which are fault tolerant, which can run concurrently (a rising issue with the current trends in processor architecture), and which provide the semantics to encapsulate compositional or algorithmic decisions atomically and consistent. For example, an algorithm may safely begin to modify a data structure, and then detect an unforeseen constellation such as an unsolvable constraint and issue a rollback, leaving the data structure without damage in its previous state. Furthermore, an STM may act as a *facade* for a durable database backend.

2.2 Persistence

In order to trace the evolution of a musical datum, the concept of persistence can be employed. Persistence means that the transactional timeline is preserved. The term can be confusing, because it is also used to describe the action of data storage on a durable medium. To avoid the ambiguity we use durability for the latter case.

Persistence has the notion of branching: A transaction may be issued from a state of the data structure at any previous stage, not just the last version, producing a new diverging data structure. Branching is a common tool in software versioning systems, allowing several developers to concurrently work on parts of the software, creating tentative branches for new features until they are ready to be merged back into the main "trunk".

Partial persistence describes a data structure which provides a read-only view of past states, while *full persistence* allows to modify past states as well [6]. *Confluent persistence* [7] allows the re-entry of structural elements or the merging of disparate branches (this operation is also known as *meld*). It is particularly useful in a musical context, as melding can be seen as a mechanism to create variations. Instead of creating a verbatim copy of an object, its reference is re-used, but tagged to individuate it so that future modifications can be applied to either version of the object. However, it has been criticised that the relationships between objects in a confluently persistent structure remain static, and an alternative approach called *retroactivity* was proposed [8], introducing the possibility to modify past states which in turn may radically affect future versions.

While confluent persistence and retroactivity have been discussed in data structure research, very little is known about actual implementations, let alone musical applications. Our contribution is to provide an actually working confluently persistent system which is backed by a durable database, thereby allowing for the growth of data structure history beyond memory capacity and for the permanent storage of musical structures.

2.3 Observers and Dataflow

While persistence algorithms provide mechanisms for the dynamic access and modification of the data structures, they do not organise the interaction between processes, between the access and modification. A meta level is needed which governs how changes in one part of the data structure can cause meaningful changes in others.

A classical concept of interaction is the Model-View-Controller (MVC) paradigm. It describes observation and reactions in a system by differentiating between these three components: A model can be most closely identified with the (musical) data structure. It is the container of "state". A view presents this state, possibly in different forms. For example, a checkbox button in a graphical user interface shows the state of a model (the model can have two logical states—on or off). It is also a controller, since when the user clicks the button, the state of the model is manipulated, which in turn is then reflected by the button displaying or hiding a checkmark symbol. In MVC, several views of the same model may coexist.

The amount of book keeping in MVC is high, and it exhibits a fragmentation of the interacting code, making it rather difficult to describe sequences of actions. It has therefore

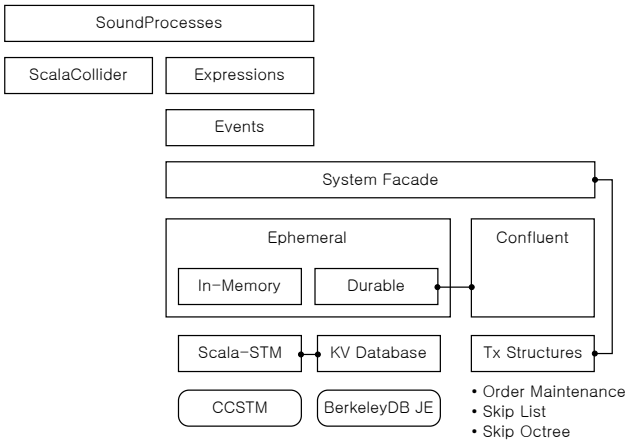

Figure 1. Schematic of the architecture of our framework.

been suggested to deprecate the MVC pattern by hiding it under the surface of a reactive data flow model [9]. The data flow model has been successfully employed in computer music systems, from Max/MSP and PD to the UGen graphs in CSound and SuperCollider. Dynamic objects are patched into each other in a declarative way. If an object earlier in the flow is updated, it automatically propagates its new state to its dependent object graph. Data flow variables may also represent ongoing calculations, and they may be used to form expressions, even if the results are not yet available.

A data flow based expression layer may address the criticism of persistence by allowing the propagation of changes in expressions from past to future states.

3. ARCHITECTURE

The components of our framework are shown in figure 1. They are written in Scala, a statically typed object-oriented and functional language running on the Java Virtual Machine. The main interface for accessing and manipulating data structures is a facade, behind which both ephemeral and confluently persistent systems may be found [1]. Algorithms and data structures developed against this facade can be parametrised in the system. For example, the confluent system uses a series of data structures which themselves have been developed against the facade. For the representation of confluent data, these are parametrised by the ephemeral durable system.

The event system is layered on top of the transactional system, and the expressions again depend on the event system. Finally, we introduce a computer music system which combines the transactional event and expression system with a realtime sound synthesis library, ScalaCollider.

3.1 System Facade

A system `S` is defined by the an opaque `ID` type, and two semi-transparent types `Var`, `Tx`. The `ID` uniquely identifies a mutable object—an object whose properties may change over time, for example a sound process which can be started and stopped and whose parameter, such as frequency, may be adjusted. In ephemeral systems, the identifier is used to establish equality even when an object is serialised and deserialised [2]. In a confluent system, an identifier is composed of a static sub-identifier and a dynamic path component which traces the movement of the object after its creation through the version graph. In a meld, an object w is re-introduced into the data structure, therefore it appears that there are now two objects w_{id_1,p_1} and w_{id_1,p_2} with the same sub-identifier id_1 but different path components p_1 and p_2.

Variables `Var` represent mutable fields and provide getter and setter methods. Whether the values are durable or kept in-memory, whether the storage is ephemeral or persistent, depends on the system type parameter. Reading and writing a variable requires a transaction `Tx` which is created by a `Cursor`. The `Cursor` in a confluent system describes how we move from vertex to vertex in the version graph.

A transaction object is also used to create new identifiers for objects (`newID`), new variables (`newVar`), and to read identifiers (`readID`) and variables (`readVar`) from disk. Figure 2 shows a linked list similar to the example used in the paper by Fiat and Kaplan [7].

```
class Nodes[S <: Sys[S]] {
  object Node {
    implicit object ser
    extends MutableSerializer[S, Node] {
      def readData(in: DataInput, _id: S#ID)
                  (implicit tx: S#Tx) = new Node {
        val id    = _id
        val value = tx.readVar[Int](id, in)
        val next  = tx.readVar[Option[Node]](id, in)
      }
    }
    def apply(init: Int)(implicit tx: S#Tx): Node =
    new Node {
      val id    = tx.newID()
      val value = tx.newVar(id, init)
      val next  = tx.newVar(id, Option.empty[Node])
    }
  }
  trait Node extends Mutable[S] {
    def value: S#Var[Int]
    def next: S#Var[Option[Node]]
    def disposeData()(implicit tx: S#Tx) {
      value.dispose()
      next.dispose()
    }
    def writeData(out: DataOutput) {
      value.write(out)
      next.write(out)
    }
  }
}
```

Figure 2. Definition of a linked list of carrying integers.

Custom types need to provide explicit serialisation code (`readData` and `writeData` in the listing). Since objects are very frequently serialised, this allows for maximum performance, at the price of more explicit code. However,

[1] Ephemeral is the antonym of persistent—an ephemeral structure does not track the history of the transaction. Once a transaction is committed the previous state of the data structure cannot be accessed any more.

[2] Serialisation is the process of encoding an object which resides in memory as a series of bytes which are then stored on a durable medium such as the hard-disk. Deserialisation is the complementary process of reading a byte series from disk and reconstructing an object in memory.

there are default serialisers for primitive types, collections and other types such as tuples. As a consequence, deserialisation is always top-down and requires knowledge about the type which is deserialised. This has implications for the event propagation, as explained in section 3.5.

The use site is much less cluttered. Figure 3 shows how a linked list of two elements is created with the above structure.

```
class Example1[S <: Sys[S]](s: S, c: Cursor[S]) {
  val nodes = new Nodes[S]
  import nodes._

  val access = s.root(_ => Option.empty[Node])

  c.step { implicit tx =>
    val w1 = Node(3)
    val w2 = Node(5)
    w1.next set Some(w2)
    access set Some(w1)
  }
}
```

Figure 3. Creating a list with elements `3` and `5`, using the previously defined `Node` structure.

The data structure has an *access* point which is initialised with a call to the system's `root` method. The cursor's `step` method issues a new transaction—passed into the closure as implicit argument `tx`—, at the end of which we may want to update the access point, in this case the head of the linked list is set to the first node `w1`.

3.2 In-Memory System

The ephemeral in-memory system is backed by a library-based interface called Scala-STM. The reference implementation CCSTM by Bronson et al. [10] is used.

3.3 Durable System

The ephemeral durable system is backed by a key-value store. This type of database behaves like a normal associative array. In the ephemeral case, the key is simply an object's `ID`, a 32-bit integer. The value is the object serialised to a byte stream. We currently use Berkeley DB Java Edition [11], a pure Java solution running as a library in the same process as the client. It supports all the ACID properties, and its transaction context is coupled with an in-memory Scala-STM transaction, so that both systems can be used conjointly.

3.4 Confluent System

The main component is the confluently persistent system. It implements the randomised compressed path method of [7], choosing suitable data structures, and modifying them to allow for quasi-retroactive operations and durability.

Path Representation Since an object which has a history is uniquely identified by a constant value and the path in the version graph leading to it, it remains to show which information is necessary for this path and how to represent it.

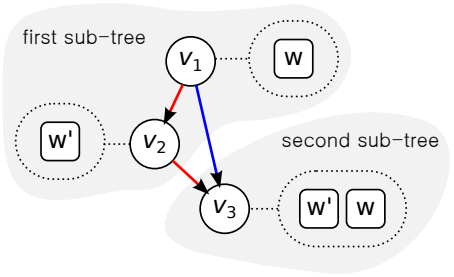

Figure 4. A meld causing the creation of a new sub-tree.

The version graph is decomposed into connected trees which are grouped by a level function. The purpose is to minimise the information necessary to uniquely identify an object in the graph. Its path component in compressed representation, $c(p)$, is a tuple $(e_1, t_1, e_2, t_2, \ldots e_i, t_i)$, where e_j denotes the version where such a sub-tree is entered, and t_j denotes the version at which the sub-path within that tree terminates. For example, let us assume versions are incrementally named v_1, v_2 etc. Let an object w be created in version v_1. Its "path" in the graph begins and ends with this single vertex, it thus has a $c(p)$ of (v_1, v_1). It is mutated in the next version step v_2, yielding a $c(p)$ of (v_1, v_2). Let us assume the data structure is a collection of objects, and in the next version step v_3 we wish to combine the first version of the object and the mutated version. As an illustration, the collection might the a group of sound producing objects, and the mutation is the adjustment of a frequency parameter. The last step then combines the adjusted sound object with its previous version, so that two instances of the same sound object are heard which are detuned in frequency.

This is illustrated in figure 4. The mutated object has thus travelled along the path v_1, v_2, v_3 (red path), while the unmutated object has travelled along the path v_1, v_3 (blue path). Looking just at the first (v_1) and last (v_3) path components, both instances are thus undistinguishable, therefore to uniquely identify them, version v_3 lies on a higher level and we enter a new sub-tree. Their compressed paths become (v_1, v_2, v_3, v_3) and (v_1, v_1, v_3, v_3) and can be distinguished [3].

The different values for a mutable object's field (that is a `Var` instance) are efficiently stored and retrieved using the randomisation method of Fiat and Kaplan. It assigns a random number to each version and produces a hash value by summing these numbers across the path. Moreover, partial hashes (partial sums) can be constructed for a fast maximum prefix search which is needed to find the most recent tree in which a given mutable object's field has been modified. In a field retrieval, the query key for the key-value store is then the tuple $(id, \sum \tilde{c}(p))$, where $\tilde{c}(p)$ is the search path with the last terminating version left out—it denotes the last sub-tree up to which is searched, and is

[3] It may sound strange that this is actually a path *compression* as the tuple size grows. But when many steps are performed which do not involve moving to a new tree (and this is the case for all operations which do not use melding), the compression ratio grows dramatically. In fact, when no melding is used at all, the compressed path is just a two element tuple independent of the number of steps performed.

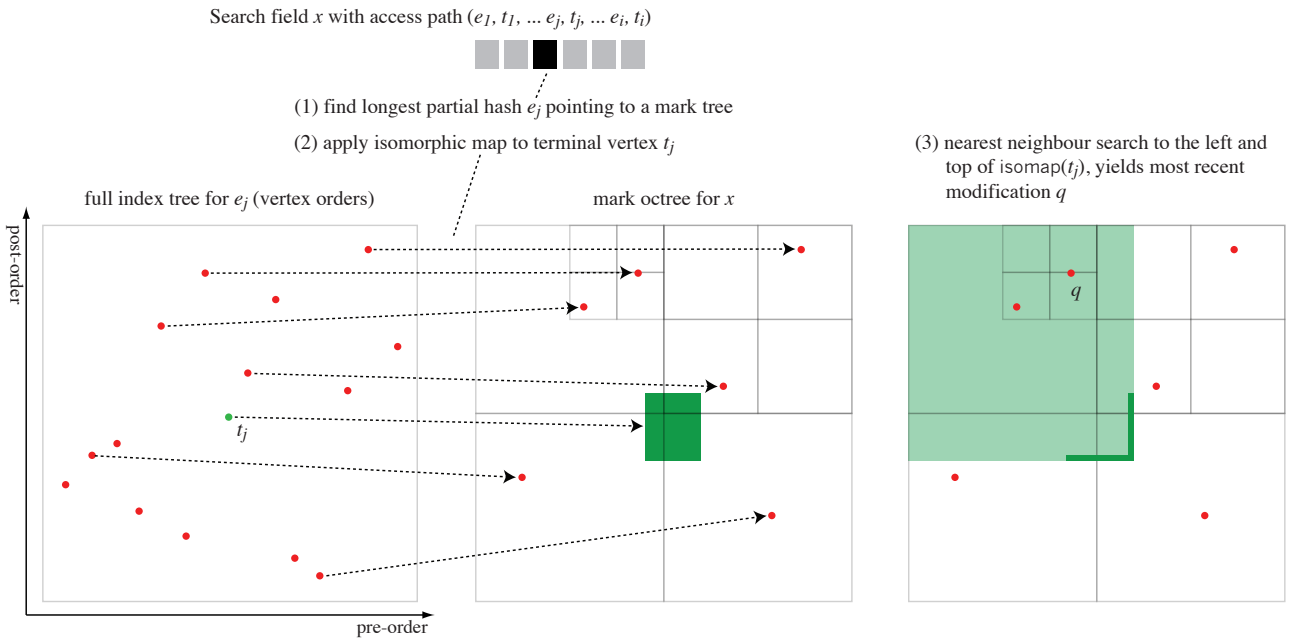

Figure 5. Overview of the retrieval algorithm for a confluently persistent variable x.

also called index of the path.

The data structure used to represent paths must be efficient with respect to splitting, concatenation, and summing any prefix of the path. A purely functional data structure which performs well in all these cases is the Finger Tree [12]. Finger Trees support the annotation of its elements with an accumulative measure. We use 64-bit integers as elements composed both of the linearly increasing version number and its associated randomised value. A measure is constructed which annotates elements both with their index position within the path as well as the sum of the randomised values from the beginning of the path up until the element. That way, elements can be randomly accessed and partial sums can be retrieved in $O(\log n)$ time.

Value Retrieval The purpose of the value retrieval is to determine which value a field had at a particular point in (transactional) time. If the frequency of a sound object was modified in versions v_1, v_4 and v_8, and we look at the object from the historic point v_6, the retrieval must report the frequency given in v_4.

Using a particular decomposition into partial hashes, the longest prefix of the search path *index* in which a field has undergone modification can be found. It points to a sub-tree, and now the version vertex within that tree must be found which corresponds to the latest modification of the field. This vertex is the nearest marked ancestor of the terminating vertex in the truncated search path. For example, if the search path $c(p)$ is $(e_1, t_1, e_2, t_2, \ldots e_i, t_i)$ and the hash based search yields longest prefix $(e_1, t_1, e_2, t_2, \ldots e_j)$ where $j \leq i$, we need to answer the question: In the sub-tree corresponding to entering vertex e_j, which is the nearest ancestor q of t_j in which the queried field has been modified?

Our answer to this question is based on the following observation: A tree can be decomposed into two total orderings, the pre-order and post-order traversal list of its vertices. A vertex v_j is ancestor of another vertex v_k, if v_j appears left to v_k in the pre-order traversal and right to it in the post-order traversal of the tree. Among the candidates, the element that is rightmost in the pre-order list is the nearest ancestor. Thus, a two-dimensional structure supporting nearest neighbour search can be used. A balanced multi-dimensional structure is the Deterministic Skip Octree [13]. Fast nearest neighbour and range search are possible because of a subsampling scheme: A helper structure, a Deterministic Skip List [14], contains the ordered leaves of the tree (the leaves carry the spatial values stored in the tree). Another helper structure is maintaining a total order of the tree's branches and leaves imposed by an in-order traversal according to the orthant indices of the tree's children. The Skip List's morphology is similar to a B$^+$Tree. As it grows, the height of the "tree" increases. The propagation of keys—the leaves' total ordering entries—up and down the list levels is used to build subsampled Octrees containing the leaves of the corresponding Skip List level.

Each value written to a mutable field within a version graph sub-tree is represented by a point stored in the Octree. The coordinates of this point are given by the position of the version vertex associated with the value in the pre-order and post-order traversal list of the sub-tree. Two problems remain: (1) The positions in the traversal lists are relative and dynamic while the Octree requires absolute and static coordinates. (2) The Octree only contains the vertices at which a particular field was modified, and for each field a separate tree must be maintained. When a version step is performed, and a vertex is inserted into the sub-tree, it is thus not feasible to update the absolute coordinates of every Octree.

A solution to (1) is to use another order maintenance structure. The elements in this order are tagged with an integer which determines their position in the order. The order between two elements can be determined by comparing

their tags. The tags remain mostly stable, only when their density becomes too high, a partial relabelling is needed. We employ the minimum tag range relabelling algorithm devised by Bender et al. [15]. When a relabelling occurs, we remove the points from the Octree, and re-insert them with their updated labels.

A solution to (2) is based on the following observation: Let the sub-tree containing all vertices—irrespective of whether assignments to a field have been made or not—be called full index tree. Let this tree be represented by the pre- and post-order traversal lists, using a tag-based order maintenance structure as described above. Consider separate pre- and post-order lists containing only the vertices where values have been written to a mutable field. These form an implicit tree called marked tree. Then, although the tags for the vertices in the marked tree and those of their counterparts in the full index tree may not be identical, their relative positions are the same. The tag list formed by the full index tree tag list after removing all non-marked vertices is thus *isomorphic* to the tag list formed by the marked tree. A search function isomap can be defined which maps a vertex from the full index tree to its theoretical position in the marked Octree.

Hence, given the terminating version t_j of the longest prefix found in the first stage of the retrieval, a nearest neighbour search in the field's marked Octree is conducted, starting at position $\mathrm{isomap}(t_j)$. The search is constrained to the orthant that contains only points smaller in the pre-order dimension and larger in the post-order dimension. The metric guiding the search is the Chebyshev distance which warrants that among the candidate points the nearest one on either axis is chosen. Function isomap has the same time cost as the actual nearest neighbour search—$O(\log n)$—, thus the nearest marked ancestor q of query vertex t_j is found in $O(\log n)$ time [4].

The algorithm can be extended to support quasi-retroactive operations as described in [16]. Where the pre- and post-order lists are manipulated ex-post, the Octree must be extended to three dimensions by including the monotonically increasing version number, so that in the ancestor search only orthants are considered which contain points that have a lower version number than the query vertex.

Figure 5 shows a schematic of the retrieval process.

3.5 Events

Events are used to model the interaction between components of the data structure. They organise how the modification of a mutable object can be registered by other dependent objects, which may in turn issue further modifications. The design specification should regard the following aspects:

No distinction between action and reaction While events should be transactionally safe, it must be possible to encapsulate any number of reaction chains within a single transaction. For example, removing a sound object from a logical group may be observed by process which alters its behaviour. It may distribute all sound objects in that group across multiple speakers, and as a result of the removal change the spatial positions of the remaining objects. This repositioning must be expressed within the same transaction as it belongs to the same logical moment in time.

Consistent view across multiple observers For instance, the removal of a sound object from a logical group is observed by two different processes X and Y. If that removal causes observer X to issue a successive removal in another group which also affects Y, Y must be able to transform the event into its particular representation, even if that yields the same object which X will remove in its reaction. A consistent representation means it is independent of the order in which the reactions of X and Y are executed.

Our event system is loosely inspired by EScala [17], another event system, albeit neither transactional nor persistent, for the Scala language. Because our system uses a top-down deserialisation approach, and because we need to be able to store event emitting and consuming objects durable in the database, an event cannot deserialise its dependents which appear opaque to it. On the other hand, events not leading to live reactions can be safely discarded, if we restrict event transforming nodes to be non-mutating (side-effect free). Events are purely data-driven, but due to the serialisation mechanism, the propagation distinguishes a forward (push) and backward (pull) phase.

Event dependents are represented by small proxies called *selectors*. We distinguish between `ReactorSelectors` which represent the non-mutating, persistent dependents, and `ObserverKeys` which through an in-memory hash table point to ephemeral, possibly mutating live reactions. In the push phase, when a `ReactorSelector` is found, the corresponding reactor is *partially* deserialised to obtain its own dependent list which is further traversed. If an `ObserverKey` is detected, the associated observer, which wraps the live reaction, is found through the hash table, and it issues the *full* deserialisation of the observed reactor. The live reactions are gathered along with these leaf reactors.

The pull phase then invokes the leaf reactors which yield the transformed events and ensure that the data structure's state is consistently seen irrespective of the reactions' execution order. Once the leaf events are calculated, the live reactions are applied. This process is illustrated in fig. 6.

3.6 Expressions

Events are commonly used for the representation of expressions in a data flow model. Expressions are similar to read-only variables in that they can be evaluated, yielding a dynamic value of an underlying type. For instance, an `Expr[Double]` evaluates to a floating point number. Expressions are composed of atoms which transform the values along the path as the expression is evaluated.

In our implementation, expressions always have a defined value. While this is not as sophisticated as fully fledged constraints programming (CP) variables (cf. [18, chap. 3]), it suffices for many scenarios. For example, a somewhat orthogonal approach to the deterministic CP is the use of bounded random variables. They can be modelled as mutable expression variables. Let *freq* be such an expres-

[4] This assumes unit pointer access costs, which does not hold when disk access from the database is involved.

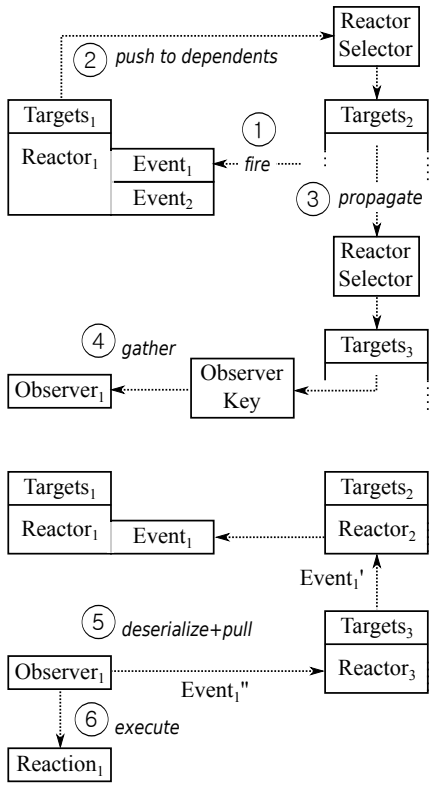

Figure 6. Event propagation. Top: push. Bottom: pull.

sion atom, denoting a frequency parameter within a given range. Operators are provided for this expression, e.g. arithmetic operators. $freq*1.4$ is composed of the numeric variable *freq* (perhaps encapsulated as random number generator), a binary operator atom $*$, and a constant atom 1.4. We may use this expression as the "value" of another expression variable, for example an adjustable control in a sound process which is interpreted as the frequency of an oscillator. Now, whenever the value stored in *freq* is mutated, a changed event is emitted, propagating through the binary operator to the variable in which the composed expression is stored, and finally reaching any live observer which may update a graphical view of the oscillator frequency, apply the change to the realtime sound synthesis process, etc.

A variable may serve different purposes, it can be a user adjustable parameter, it can model a random number generator [5] , but also hold sampled live sensor or audio analysis data (i.e. in our SuperCollider based system it could be set by a sampled unit generator, such as `Pitch` or `Loudness`).

4. TOWARDS A COMPUTER MUSIC SYSTEM

As indicated in fig. 1, a preliminary computer music system, SoundProcesses, has been set up, bridging the previously described framework with ScalaCollider [19], a Scala client for the SuperCollider server. Since SuperCollider uses UGen graphs which have a structure similar to expression trees, it is straight forward to reuse the unary and binary UGen operators for client side expressions on integer or floating point numbers. In future versions, the UGen graphs themselves may be seen as expressions, so that a sound process can be specified as a transformation of the UGen graph of another sound process.

4.1 Aural Presentation

A previous version of SoundProcesses has been successfully used in a range of applications, from live electronic performance to generative sound installations. While it was already transactionally safe, it did not implement a persistent layer, and it conflated data structure (the model of sound processes) with presentation (the actual control of the sound synthesis server). In the new implementation, the concept of *aural presentation* is introduced which is the aural "view" of a model. It will allow to manipulate the structure even when there is no realtime sound synthesis server present (e.g. in offline editing), and different "transports" may simultaneously present different versions of the sound process. The distinction is also necessary to establish the bi-temporal layer—something which we have omitted in this paper—, where editing cursor (version or transaction pointer) and playhead cursor (valid time pointer) diverge [6] .

An `AuralPresentation` observes a `Group` of sound `Procs` (*descriptions* of sound processes, i.e. data structures). The group emits events when processes are added or removed. When a process is added, the aural presentation creates an ephemeral view of the process, for example it may send the UGen graph to the SuperCollider server, run a node on the server, adjust the node's control parameters on the server, etc.

Since a transaction may fail and be rolled back, it must however not issue irreversible actions until it is certain that the transaction will successfully complete. SuperCollider is controlled through Open Sound Control (OSC) messages, and once a message is sent out, there is no way to revoke it. On the other hand, it does have a model of atomicity, wherein messages contained within the same OSC bundle become effective at the same logical time. The aural presentation maintains proxies for server side objects which are updated during the transaction. When the transaction is committed, the updates of these proxies are translated into appropriate OSC messages which are then sent out.

4.2 Example

Figure 7 gives an impression of how the code for a simple recursive duplication of a sound process currently looks like. Object `ExprImplicits` provides infra structure for the expression system. The data structure's root is a group which is used for the aural presentation. An expression variable initially holding the numeric value of 50.0 is created. In the next version (cursor step), a new process is

[5] The generator expression then has an `update` trigger method which calculates a new pseudo random number based on an internally stored transactional and persistent "seed" value, and fires the corresponding changed event.

[6] We are currently modelling bi-temporality by providing an additional *valid time* cursor for reading and writing expressions. Time can be represented as another numeric expression (e.g. frames relative to the start of a transport) and for efficient lookup the temporal occurrences are evaluated and stored in a caching structure such as a binary search tree.

instantiated. It has a parameter `freq` which is set to the expression variable created in the previous version. The `graph` parameter holds a so-called `SynthGraph` object, an unexpanded UGen graph (cf. [19]). Here a set of harmonically spaced sine oscillators with slight random detuning is used. The fundamental frequency is control by the process' `freq` parameter. The process is added to the group, so it becomes audible as soon as the aural presentation has launched the SuperCollider server.

The call `p.asAccess` creates an auxiliary entry to the data structure, so we do not need to find the process in the actual data structure's root (the group), but may access it directly in later transactions. As there is no thorough cursor management in place yet, the assignment of `v1` saves the most recent path in the version graph which will be used in the meld. We intentionally use the same version graph as depicted in figure 4, so the process just created can be identified with object w in version v_1.

Next we modify the process by reassigning the frequency parameter, now being formed of a binary operation (multiplication) of the expression variable. The event system will propagate this change, and the aural presentation changes the frequency value of the synthesiser to 70.0 Hz.

In version v_3 the old version of the process is melded into the group, thus there are two sound generators now which use the same DSP configuration, but are detuned against each other. Finally, in version v_4, the expression variable is changed, so that the frequencies of the two variants of the process become 56.0 and 40.0 Hz respectively.

4.3 Melding and Events

The operation of v_4 introduces a grave problem: If expressions are realised in the regular confluent persistence semantics, the meld also "duplicates" the frequency expression variable. In v_2 the process' frequency parameter is removed as dependent of the expression variable, and instead the binary operator is introduced. The expression variable dependencies are updated for path v_0, v_2. In the meld of v_3, when the process variant is added to the group, the event dependency graph is reengaged for the newly observed object. However, it will lead to an expression variable with path v_0, v_0, v_3, v_3. Depending on whether one looks at the expression variable from the perspective of either process variant, it now has different event targets, and a modification of its value will only propagate to either of the process variants.

Expressions must therefore be treated as partially persistent—their identity (path component) should never be split as part of a meld operation. They form the hinge between the confluent sound processes and ephemeral views or aural presentations. At this stage, we are still investigating the transition from confluently to partially persistent objects and vice versa. It seems sufficient to carry around the confluent path representations and simply fall back to partial paths internally in the access and update of partial variables. A partial path is just the tuple (e_1, t_i), where e_1 is the seminal version of the persistent object and t_i is the terminal version of the current access path.

```
type S = Confluent

class Example2(implicit s: S, c: Cursor[S]) {
  val imp = new ExprImplicits[S]
  import imp._

  val group = s.root(newGroup()(_))
  AuralPresentation.run[S](group)

  val freq = c.step { implicit tx => // v0
    exprVar(50.0).asAccess
  }

  val w = c.step { implicit tx =>    // v1
    val p   = newProc()
    p.freq  = freq
    p.graph = {
      val m = Mix.tabulate(20) { i =>
        FSinOsc.ar("freq".kr * (i + 1)) *
        (LFNoise1.kr(Seq(Rand(2, 10), Rand(2, 10)))
        * 0.02).max(0)
      }
      Out.ar(0, m)
    }
    group add p
    p.asAccess
  }
  val v1 = c.step(_.inputAccess)

  sleep(4.0)
  c.step { implicit tx =>            // v2
    w.freq = freq * 1.4
  }

  sleep(4.0)
  c.step { implicit tx =>            // v3
    group.add(w meld v1)
  }

  sleep(4.0)
  c.step { implicit tx =>            // v4
    freq.get set 40.0
  }
}
```

Figure 7. Re-entry of a sound process, and reactive update of an expression.

5. CONCLUSIONS

We have presented a framework which implements data structures to register the transactional timelines which may represent the process in which a computer music composition is created or generatively updated. It allows a composition to access its own evolution, a basic form of which is the re-introduction (melding) of a diverging version of a sound object into the piece's structure.

To the best of our knowledge, this is the first implementation of confluent persistence, paired with a database backend. Previous attempts have only realised in-memory partial persistence [20]. Furthermore, a transactional and persistent event propagation system has been designed. As noted in the previous section, the transition from confluently persistent objects (such as meldable sound processes) to partially persistent expressions is still not fully elaborated.

The most important task now is to put the framework into production, and design a simple system based on SoundProcesses which allows it to be used in different contexts and by different composers. While there have not been any issues with our preliminary test cases, it remains to be seen how the performance of the system scales in different scenarios. Most operations exhibit logarithmic slow-down in a RAM model, but may indeed have polylogarithmic costs due to the database backend. On the other hand, there are various occasions where cache mechanisms are employed, and it may thus be more a matter of profiling the patterns emerging from the existing or prospective cache strategies. An advantage of the aural presentation approach is that the sound synthesis server runs independently, weakening realtime constraints on the framework itself.

6. REFERENCES

[1] H. Honing, "Issues on the representation of time and structure in music," *Contemporary Music Review*, vol. 9, no. 1, pp. 221–238, 1993.

[2] A. Di Scipio, "Compositional Models in Xenakis's Electroacoustic Music," *Perspectives of New Music*, vol. 36, no. 2, pp. 201–243, 1998.

[3] C. Roads, *The computer music tutorial*. Cambridge, MA: The MIT Press, 1996.

[4] R. Snodgrass and I. Ahn, "A taxonomy of time databases," *ACM SIGMOD Record*, vol. 14, no. 4, pp. 236–246, May 1985.

[5] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," *ACM Computing Surveys (CSUR)*, vol. 15, no. 4, pp. 287–317, 1983.

[6] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan, "Making data structures persistent," *Journal of Computer and System Sciences*, vol. 38, no. 1, pp. 86–124, Feb 1989.

[7] A. Fiat and H. Kaplan, "Making data structures confluently persistent," *Journal of Algorithms*, vol. 48, no. 1, pp. 16–58, 2003.

[8] E. D. Demaine, J. Iacono, and S. Langerman, "Retroactive data structures," *ACM Transactions on Algorithms (TALG)*, vol. 3, no. 2, pp. 13:1–13:20, 2007.

[9] I. Maier, T. Rompf, and M. Odersky, "Deprecating the Observer pattern," *Technical Report EPFL-REPORT-148043. Ecole Polytechnique Fédérale de Lausanne*, 2010.

[10] N. Bronson, H. Chafi, and K. Olukotun, "CCSTM: A library-based STM for Scala," in *Proceedings of the First Scala Workshop*, 2010.

[11] "Berkeley DB Java Edition Architecture," *An Oracle White Paper*, September 2006.

[12] R. Hinze and R. Paterson, "Finger trees: a simple general-purpose data structure," *Journal of Functional Programming*, vol. 16, no. 2, pp. 197–217, 2006.

[13] D. Eppstein, M. T. Goodrich, and J. Z. Sun, "The Skip Quadtree: A Simple Dynamic Data Structure for Multidimensional Data," in *Proceedings of the twenty-first annual symposium on Computational geometry*. ACM, 2005, pp. 296–305.

[14] T. Papadakis, "Skip Lists and Probabilistic Analysis of Algorithms," Ph.D. dissertation, University of Waterloo, Waterloo, Ontario, 1993.

[15] M. A. Bender, R. Cole, E. D. Demaine, M. Farach-Colton, and J. Zito, "Two simplified algorithms for maintaining order in a list," *Algorithms—ESA 2002*, pp. 219–223, 2002.

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

[17] V. Gasiunas, L. Satabin, M. Mezini, A. Núñez, and J. Noyé, "EScala: modular event-driven object interactions in Scala," in *Proceedings of the tenth international conference on Aspect-oriented software development*. ACM, 2011, pp. 227–240.

[18] T. Anders, "Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System," Ph.D. dissertation, School of Music & Sonic Arts, Queen's University, Belfast, 2007.

[19] H. H. Rutz, "Rethinking the SuperCollider Client...," in *Proceedings of the SuperCollider Symposium*, Berlin, 2010.

[20] F. Pluquet, S. Langerman, and R. Wuyts, "Executing code in the past: efficient in-memory object graph versioning," in *ACM SIGPLAN Notices*, vol. 44, no. 10, 2009, pp. 391–408.