ON THE TRACEABILITY OF THE COMPOSITIONAL PROCESS

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

Eduardo Miranda
University of Plymouth
Interdisciplinary Centre for
Computer Music Research (ICCMR)
eduardo.miranda@plymouth.ac.uk

Gerhard Eckel
University of Music
and Performing Arts Graz
Institute of Electronic Music
and Acoustics (IEM)
eckel@iem.at

ABSTRACT

Composition is viewed as a process that has its own temporal dimension. This process can sometimes be highly non-linear, sometimes is carried out in real-time during a performance. A model is proposed that unifies the creational and the performance time and that traces the history of the creation of a piece. This model is based on a transformation that enhances data structures to become persistent. Confluent persistence allows navigation to any previous version of a piece, to create version branches at any point, and to combine different versions with each other. This concept is tuned to integrate two important aspects, retroactivity and multiplicities. Three representative problems are posed: How to define dependancies on entities that change over time, how to introduce changes ex-post that affect future versions, and how to continue working on parallel versions of a piece. Solutions based on our test implementation in the Scala language are presented. Our approach opens new possibilities in the area of music analysis and can conflate disparate notions of composition such as tape composition, interactive sound installation, and live improvisation. They can be represented by the same data structure and both offline and realtime manipulations happen within the same transactional model.

1. CONCEPTUAL FOUNDATION

1.1 The Double Nature of Composition

Time, along with space the fundamental parameter of composing pieces of music and sound art, appears in various forms. A fundamental distinction can be drawn between the *creational* time t_K – the time in which a composer creates or manipulates a piece – and *performance* time t_P – the time in which a piece is presented to a listener. One feels reminded of the double nature of the term *composition*, as pointed out by Koenig: «By musical composition we generally understand the production of an instrumental score or a tape of electronic music. However, we also understand composition as the result of composing [...] (we say for instance: "I have heard a composition by composer X").» [1, p. 191] The hermetic view that the «concept of composition is accordingly closed with regard to the result, but open with regard to the making of a composition» [1, ibid.], however is dropped in favour of one where the electroacoustic composer is regarded as the «first listener» [1], often being able to "perform" the piece in total or part while working on it, or in fact performing it in a live improvisation where part of the work is composed (put together) while being performed, not necessarily arriving at one pre-defined result.

A special case is added by the medium of sound installation, where the "piece" often does not have a beginning or ending, and in which the listener chooses the time span of exposure to the sounds, sometimes even influencing the piece in an interactive way. It is therefore useful to depart from a perspective where the process of composition *terminates* in a fixed composition, but rather to consider indeterminacy as an essential ingredient, and therefore to look out for ways in which indeterminacy can be represented and manipulated in a composition system. Tentatively, we further divide elements in t_P into those forming a virtual (or prospective) structure and into those forming several actual realisations of that structure.

1.2 Databases

An interesting taxonomy has been developed in research on *database* systems: In a bi-temporal database, two timelines are distinguished: The *valid* time defines the time in which a database entry has existence in "reality", when it «accurately modeled reality» [3]. On the other hand, someone is maintaining and editing the database, performing operations on the transactional level, where time means «when an event is recorded in the database» [4]. The two timelines are often seen as orthogonal to each other, and common queries are punctiform with regard to transactional time and interval based with regard to valid time (cf. [5]). It follows that, if a composition is con-

[1] The term is deliberately taken from Daniel Charles who demands «a music that takes care to consider the composer not as the organizer of a technological ritual but, more modestly, as the first listener». [2, p. 152]

sidered being a kind of database, the composer takes the role of the operator on the database, and transactional time corresponds with t_{K}. The entries in the database, the elements from which the music is constructed, form one or more valid timeline fragments in virtual t_{P} and eventually become one particular actual timeline in t_{P} per performance.

1.3 Persistence

The extension of the valid by the transactional time layer introduces the history of the creation of a work. In the taxonomy of Driscoll et al., we call *ephemeral* a data structure which is agnostic of its history, so that any modification to it would let the previous state fall into oblivion. On the other hand, enhancing an ephemeral structure so that its previous states are still accessible, makes it become persistent [6]. The distinct variants of an ephemeral structure we call "versions", and the versions form vertices in a directed graph such that one vertex v_j points to another v_k if v_k was created by applying some modification to the ephemeral structure in version v_j.

In a linear perspective, each transaction corresponds with a new version of the database. However, in a non-linear perspective, one could depart from any previous version and branch off. Finally one could even create a version by combining two previous versions. In the first case, the graph is a linear path, and the enhanced structure is called *partially* persistent, meaning that «all versions can be accessed but only the newest version can be modified». The second case produces a graph which is a tree, and we speak of full persistence, where «every version can be both accessed and modified». In the last case the enhancement which uses «an ephemeral data structure that supports an update in which two different versions are combined» is called confluent persistence, and the versions have the most general form of a directed acyclic graph (DAG) [6].

We will use persistence not only to model the evolution of a piece in t_{K}, but also to use version branching and melding as a *joint* between virtual elements and one or more actual realisations in t_{P}. This way we unify the compositional process and the performance of pieces.

A critique of the persistence approach comes from Demaine et al. who state that, since versions are never overwritten and versions can only depend on previous versions, «the dependence relationship between two versions never changes. [...] Thus, the persistence paradigm is [...] inappropriate for when changes must be made directly to the past state of the structure.» [7] Instead they propose «retroactive data structures» as a new approach that can incorporate deliberate manipulations of past states of a data structure. Unlike persistence for which general transformations have been proposed, retroactivity requires special solutions for each particular data structure. In section 2.5 we will face a problem that seemingly calls for some kind of retroactivity, and we will see that it can well be solved within the persistence paradigm.

1.4 Multiplicities

If now the compositional process is seen as a sequence of decision-making or actualisation, we may integrate indeterminacy into this model. Indeterminacy can be attributed to three sources: Chance operations, interactive sensorial input, and generative (self-modifying or memorising) structures. Indeterminacy can be subsumed under other forms of multiplicities, namely the exploratory behaviour of the composer who concurrently or successively develops different versions of (parts of) a piece, and scale where different versions are developed for different contexts, e.g. modes of spatialisation. In total this leads to five types of multiplicities. The task is then to elaborate appropriate models for the virtual sources of multiplicities, for example a model of chance operations, a model of interactive input, etc. Although this is beyond the scope of this paper, we will use a simple placeholder for temporal values which are unknown in a version to show that our general model can indeed be extended to represent multiplicities.

2. IMPLEMENTATION

2.1 An Overview of Confluent Persistence

Fiat and Kaplan [8] have developed general algorithms for turning any ephemeral linked data structure into its confluently persistent counterpart. Our contribution is to apply this framework to the representation of temporal objects, extending it in several ways. Before we describe problem cases and their solution, it is therefore necessary to give a brief overview of this framework.

The ephemeral data structure is considered to be composed of any number of nodes each of which can have "data" and "pointer" fields. The pointer fields are used to link nodes together, while the data fields hold primitive values. Instead of distinguishing data and pointer fields, we prefer to speak about mutable fields used to store *immutable* values and mutable fields used to refer to other *mutable* objects.

Each ephemeral node is transformed into a "fat" node which contains information about all its states through different versions, using some kind of dictionary for each field. To distinguish which version we are looking at, the notion of a node pedigree is introduced which basically is a sequence of version identifiers (or vertices), starting from the version in which a node was created (the seminal version), and carrying successively the identifiers of the versions through which the node was brought to the currently accessed version. The identifier of a fat node thus is the tuple composed of the node pedigree and a reference to the fat node structure.

To access a field in a particular version of a fat node, each ephemeral field is replaced by a fat field which consists of a *search trie* [2] that carries all values that have ever been assigned to that field, stored in the leaves of the trie, and the paths into the trie being the so-called assignment pedigrees, again a sequence of version identifiers. In the case of pointer fields – references to mutable objects –, node identifiers are stored in the trie, and in the retrieval a transformation called «Pedigree Prefix Substitution» is applied to update the node identifier so that it becomes a valid and unique access identifier within the current path in the version DAG.

The notion of pedigrees allows for the appearance of an ephemeral node more than once within the same version, while maintaining correct access to each element. The operation by which a node is re-introduced into a version is called a "meld", and it allows for example to catenate a linked list to itself or to an older version of itself.

Our implementation uses Fiat and Kaplan's *compressed-path* representation of pedigrees. It is based on the observation that there are often long linear sequences in the version graph which produce weak performance when using a full-path pedigree representation. In the full-path representation, the trie keys grow linearly with the number of versions. In the compressed-path method, the graph is split into disjoint (sub-)trees, each of which has an associated level ℓ, can only be entered at most once per path and will be represented by two symbols – the identifier of the version at which one enters the subtree and the identifier of the version from which one leaves the subtree (or the terminal version if the path ends in this subtree). A new subtree with an incremented level needs to be created when a meld operation is based on elements from versions of the same tree level. The so-called index $\tilde{c}(p)$ of the compressed path $c(p)$ is used now as key into the search trie of the fat fields. It contains all elements of $c(p)$ but the last, the particular version vertex inside one subtree. The value stored in the trie is a data structure that contains all the mappings from the target vertices of assignments (last elements of compressed paths) to the assigned values. Given a query key, this structure can find the nearest ancestor vertex in the subtree.

For the tries, we employ the lexicographic *splay* tree of Sleator and Tarjan [9], for the target vertex mapping we use a plain list along with a total ordering of the vertices imposed by the pre-order and post-order of the subtree, as suggested by Dietz [10] [3], although more sophisticated and better performing data structures are known (cf. [11]).

[2] A trie, also called prefix tree, is an associative data structure where the key consists of a sequence of elements. To find a value in the trie, the first element of the key is compared to the root node and the according branch is taken, then the second element is compared to the node at the second level, and so forth.

[3] Briefly, 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.

2.2 Posing a Problem

We will introduce our approach by posing a simple problem, illustrated in Figure 1, that is to be solved.

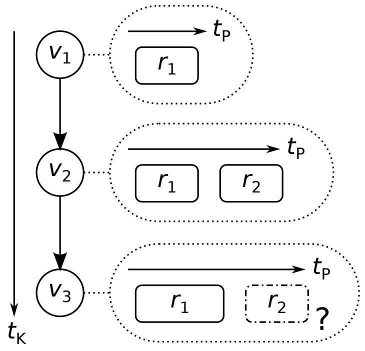

Figure 1. First problem: How to define the dependancy of region r_2 on r_1 such that the former automatically moves along with the latter in future versions?

Bi-temporality is modeled by associating each version vertex v_j with a point on the creational timeline (implicit), and the performance timeline t_P is expressed in the ephemeral data structure such that objects in time are associated with a time interval in t_P. Let the basic class of such objects be (fat) regions, denoted by r_j. A region can be anything from an audio file snippet to a code block that generates synthesised sound. Let $i(r_j)$ be the *interval* of a region, specified by a beginning time *start* and a duration *dur*, such that $i(r_j) = (start, dur)$. These points in time we call *periods*.

There are three operations performed by the composer. In v_1, a new region r_1 is created, forming the initial element of the performance timeline. The second operation, forming the next version v_2, adds a second region r_2 such that $i(r_2)$ starts at a fixed offset after $i(r_1)$ stops. Finally, the composer decides that r_1 should last longer and adjusts the duration of its interval accordingly. The problem is to define the relationship between r_1 and r_2 such that under the modification of r_1, we preserve the intention of r_2 following r_1.

We assume a set of arithmetic operations on intervals and periods, such as addition [4]. In v_2 we would say that $i(r_2)$ is created by an expression such as $(start = stop(i(r_1)) + p_o, dur = p_d)$ where p_o is some period offset and p_d some period duration. However, the manifest idea of applying the confluent persistence technique to either *resolve* the value of the fat interval field of r_1 at version v_2 or to create a fat pointer *reference* entry to it at version v_2 would create the fixed "dependence relationship" that was criticised by the retroactive approach – $i(r_2)$ would depend on $i(r_1)$ assigned in the version that is closest ancestor of v_2. What we want instead is to depend on this interval *no matter* at which version of $i(r_1)$ we are looking. The solution is to use a kind of dynamic reference.

[4] E.g., we simply define an interval's *stop* as *start* + *dur*.

Before we give this solution, we present our testing framework.

2.3 The Testing Framework

For our implementation, we use the Scala programming language [12]. Scala combines object-oriented and functional approaches and has a rich type system with single class inheritance and multiple mixins called *traits*. A trait can declare a set of abstract methods, but can also provide concrete definitions. Although Scala is a compiled language, we use a read-eval-print-loop (REPL) that allows one to create version vertices and navigate between them step-by-step. We also anticipate that in a composition environment based on this framework, the composer would typically create structures in a REPL. Furthermore, we provide access to the persistence sensitive environment in the form of an internal domain specific language extension which is realised by a combination of methods imported into the REPL scope and so-called implicit conversions, a language construct of Scala that can be used to seemingly enrich existing classes with new methods. For example, we add a method `secs` to the floating point class `Double` to create period literals. Figure 2 shows an overview of the classes involved.

The environment maintains two access paths into the version graph, one for reading and one for writing. When creating a new version, referencing and accessing existing objects involves the reading path which denotes the version we are departing from, and assignments are made using the writing path which denotes the newly created version. A single version step is performed by method `t[T](thunk: => T): T` which takes an argument *thunk* – a parameterless function with result type `T` – and evaluates it in a context where the read access corresponds to the current version, and the write access corresponds to a version newly derived from the current version. The first step in figure 1 becomes:

```
val r1 = t { region("r1", 0.secs :< 3.secs) }
```

Note how the interval literal is constructed by taking a *start* and a *dur* period and catenating them with the `:<` operator. The result of this operation is stored in value `r1`. It is a special region *handle* that the environment automatically converts to a region *identifier* when used. This saves us from explicitly updating each access identifier when navigating through the version graph.

2.4 Solution to the Problem

Assuming that method `interval` on the region object returns a reference with the desired semantics as requested in the conclusion of section 2.2, the code for versions v_2 and v_3 becomes:

```
val r2 = t { region("r2", (r1.interval.stop +
    2.secs) :< 5.secs) }
t { r1.interval = 0.secs :< 7.secs }
```

These semantics are achieved by constructing an `IntervalProxy` that wraps the underlying fat interval field in r_1. This proxy delegates the interval methods by using a special `access` method. When arithmetics are performed on the proxy's `start` or `dur` fields, they are wrapped in special `PeriodExpr` objects.

The `access` call performs a pedigree prefix substitution as in the pointer retrieval of [8], but *prior* accessing the fat field. As a consequence, the proxy acknowledges all modifications made to the field between the creation of the proxy and the current version. We therefore call this behaviour "fluent". In order to access a region's interval without the fluent behaviour, the proxy is fixed via `r1.interval.fixed` (cf. fig. 2). Note that linearity in t_{K} is maintained, so $i(r_2)$ in v_2 is still referring to $i(r_1)$ in version v_1!

2.5 Retroactive Operations

In the second problem, the approach of section 2.4 does not help, as we are now faced with version branching. The problem is depicted in figure 3.

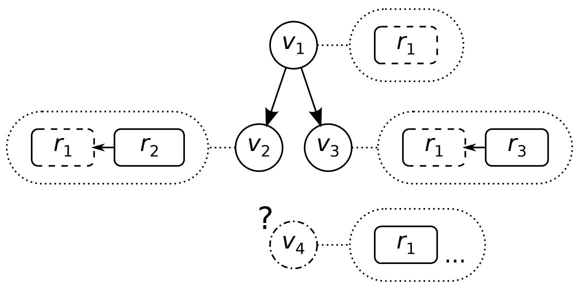

Figure 3. The second problem addresses changes that need to affect more than one future version.

Here the composer decided to begin with a region r_1 but leaves its duration subject to a later decision. Next, a version v_2 is produced by adding another region to start after region r_1 stops, similar to the previous example. The dependency of the regions is indicated by the arrow from r_2 to r_1. After completing version v_2, the composer tries out a different variant, starting over from version v_1, the result of which is version v_3. The finishing task, deciding on the final interval of r_1, seemingly calls for a retroactive operation – a correction of v_1. However, throughout our framework we pursue the preservation of causality, that is, the original states of $v_1 \dots v_3$ must still be accessible. The solution is to insert a new vertex v_4 between v_1 and its former children, as shown in figure 4(a).

The data structure maintained for the nearest ancestor search within a subtree – and until now we have only dealt with graphs consisting of a single tree – can be tuned for such a quasi-retroactive insertion. Method `retroc` inserts a new version vertex (here v_4) right after its parent vertex (here v_1) in the pre-order traversal list, and right before it in the post-order traversal list. In this case, this yields a pre-order of $\langle v_1, \boldsymbol{v_4}, v_2, v_3 \rangle$ and a post-order of $\langle v_2, v_3, \boldsymbol{v_4}, v_1 \rangle$, satisfying the condition of v_4 being the closest ancestor

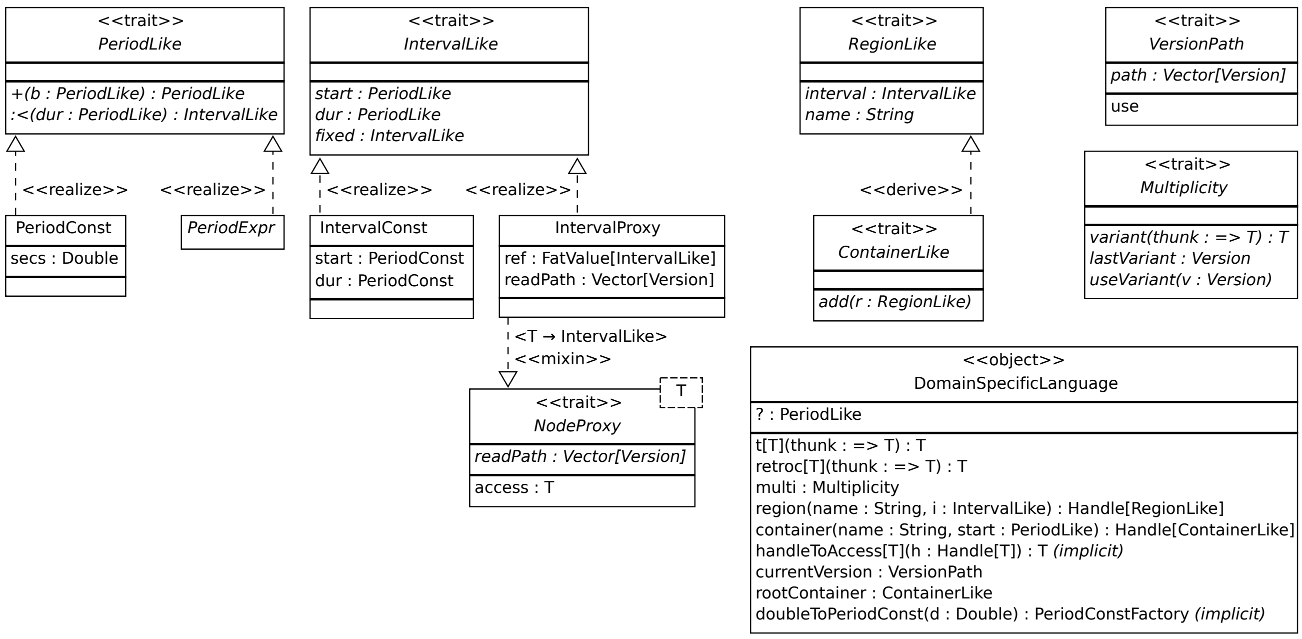

Figure 2. Class Diagram. Depicted are only those classes, attributes and methods referred to in this paper.

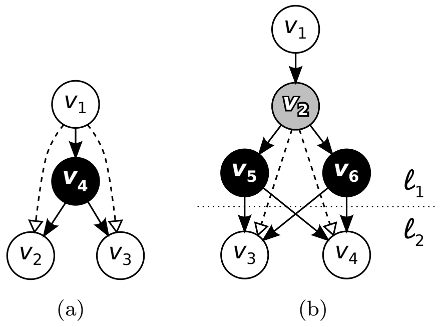

Figure 4. Retroactive vertices. (a) Single "corrective" operation. (b) Incorporating multiplicities.

of v_2 and of v_3 (cf. footnote 3). The corresponding code is:

```
val r1 = t { region("r1", 0.secs :< ?) }
val v1 = currentVersion
val r2 = t { region("r2", (r1.interval.stop +
    2.secs) :< 5.secs) }
v1.use
val r3 = t { region("r3", (r1.interval.stop +
    3.secs) :< 7.secs) }
v1.use // parent of the retroactive vertex
retroc { r1.interval = 0.secs :< 4.secs }
```

We use `?` as a placeholder for an unspecified period. The `currentVersion` method is used to capture the currently accessed version path. Navigation back to a particular version is achieved by calling `use` on a version path.

It has not been explained yet how the linearity of t_K is preserved, so that the original states of versions v_2 and v_3 are not lost. This is achieved by conditioning the ancestor lookup using the monotonically increasing vertex indices: When looking up a target vertex v_k, only vertices $v_j, j \leq k$ are considered. Therefore, v_4 becomes effective only after creating further descendants from the graph's leaves. For example, if v_5 is created from v_2, the modifications of v_4 will be effective in v_5 (since $4 \leq 5$), but not in v_2 (since $4 > 2$), preserving the original state of v_2. This conditional behaviour is indicated in figure 4(a): In v_2 and v_3 the dashed arrows are followed, while in descendants of v_2 and v_3 the solid arrows are effective.

Going back to the conceptual layout, intuitively one could think of using *repeated* retroactive insertions to represent the various outcomes of a multiplicity. For instance, if $\overset{*}{v_5}$ was a new correction to $i(r_1)$, it would be inserted as a retroactive child vertex to v_4. However, the composer would not be able to freely *switch* between these two variants for $i(r_1)$ at a later point, precisely because we enforced the linearity in t_K.

We introduce another method `multi` which is dedicated to this problem. `multi` can only be executed on leaves of the graph, because it enforces a successive tree split. Figure 4(b) illustrates this: Initially, `multi` creates a *neutral* vertex v_2 which functions as a common ancestor for any future outcomes of the multiplicity. The versions v_3 and v_4 inserted after the multiplicity are enforced to start new subtrees at level ℓ_2. All vertices belonging to the multiplicity, located at the smaller tree level ℓ_1, will be explicitly included in the compressed path-representations as exiting vertices and can thus be seen as mutual switches. For instance, if we wish to access version v_3 incorporating variant v_5, the compressed path would be $\langle v_1, \boldsymbol{v_5}, v_3, v_3 \rangle$, if variant v_6 was desired, the path would be $\langle v_1, \boldsymbol{v_6}, v_3, v_3 \rangle$.

2.6 Melding and Parallel Motion of Versions

The enforcement of tree splitting in `multi` makes one think of the original meld operation. Indeed the vertices of the versions succeeding a multiplicity may have an indegree of > 1 (the indegree is the number of variants realised) which is also characteristic of a version meld. The difference is, that in the original confluent persistence framework, the set of access pointers to the data structure remains the same. With the use of multiplicities we are facilitating what we call "parallel motion" in the version graph so that there are different access paths to a particular vertex, carrying the information about which variant of each multiplicity is "active". This is clarified by a final example, shown in figure 5.

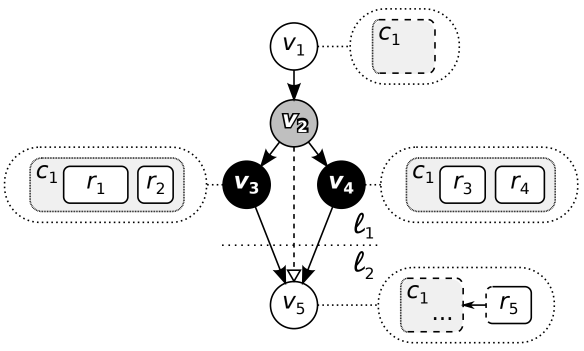

Figure 5. The third problem demands a facility to continue in a piece without settling on one particular version (v_3 versus v_4).

Here, the composer has planned a section for live improvisation and created a *container* c_1 for it in the first version. A multiplicity (neutral vertex v_2) is created to host the different improvisations, so one can later switch forth and back between them. The two variants v_3 and v_4 are simply modeled by adding regions to the container, as the framework really is agnostic to whether an operation is carried out offline or in realtime. As the composition finally carries on in v_5, this version automatically has a tree level incremented from ℓ_1 to ℓ_2. The following code simulates this:

```
val c1 = t { container("c1") }
val m = multi
c1.use
val (r1, r2) = m.variant {
    (region("r1", 0.secs :< 4.secs),
     region("r2", 5.secs :< 2.secs)) }
val v3 = m.lastVariant
val (r3, r4) = m.variant {
    (region("r3", 0.secs :< 2.5.secs),
     region("r4", 3.5.secs :< 3.secs)) }
val v4 = m.lastVariant
rootContainer.use
val r5 = t { region("r5", (c1.interval.stop +
    2.secs) :< 4.secs) }
m.useVariant(v3)
r5.interval.fixed // result: 9.secs :< 4.secs
m.useVariant(v4)
r5.interval.fixed // result: 8.5.secs :< 4.secs
```

As can be seen, method `variant { }` creates a new variant transaction, while `useVariant` updates the access path to include a particular variant. Since independent regions are created in the two variants, they cannot be used as references for v_5. This reference problem is solved by introducing the *container* object. Implicitly, in all the previous examples, the regions had been added to the default `rootContainer`. We explicitly revert to it for region r_5 by calling `rootContainer.use`.

The neutral vertex v_2 gains additional significance here: As r_5 is created, it is added to the root container. Assuming that containers are modeled using a list of the contained objects along with a field for the size of this list, assignments with compressed path $\langle v_1, \boldsymbol{v_2}, v_5, v_5\rangle$ are produced in the fat root container. If now variant v_3 is activated, the current access path becomes $\langle v_1, \boldsymbol{v_3}, v_5, v_5\rangle$. Consequently, a query for the number of objects in the root container would terminate in the trie at v_1, producing the wrong result, since c_1 was the only container in v_1.

This last problem is solved by enhancing the maximum prefix search in the trie such that if a vertex (here v_3) is not found and this vertex belongs to a multiplicity, it is replaced by the corresponding neutral vertex (here v_2) and the last splaying is repeated.

3. CONCLUSION

We have modeled and an implemented the music composition process as a confluently persistent data structure where the version DAG forms the creational timeline t_K, and the structure itself contains temporal objects relating to performance time t_P. We have enhanced this structure with two new operations `retroc` to incorporate quasi-retroactive decision making and `multi` to integrate realisation variants of the piece. The apriority of Allombert et al. [13] – "1. The compositional process: the composer builds his interactive score [...] 2. The performance process: the interactive score is no more edited" – becomes meaningless, as the realisations become part of "the piece", they re-build it.

The framework remains to be tested in a real-world and real-time application. For this, an efficient scheduler representation of the temporal objects is needed. Changes induced by retroactivity and switching between variants of a multiplicity need to be propagated (e.g., in some form of publisher-subscriber pattern), and cases where queries become invalid must be handled. This question of inconsistency has been investigated by Acar et al. [14].

4. REFERENCES

[1] G. M. Koenig, *Ästhetische Praxis*, vol. 3 of *Texte zur Musik*, ch. Kompositionsprozesse (1978),

pp. 191–210. Saarbrücken: PFAU Verlag, 1993.

[2] D. Charles, "Entr'acte: "Formal" or "Informal" Music?," *The Musical Quarterly*, vol. 51, pp. 144–165, Jan 1965.

[3] R. Snodgrass and I. Ahn, "A taxonomy of time databases," *ACM SIGMOD Record*, vol. 14, pp. 236–246, May 1985.

[4] G. Copeland and D. Maier, "Making smalltalk a database system," *ACM SIGMOD Record*, vol. 14, pp. 316–325, June 1984.

[5] B. Salzberg and V. J. Tsotras, "Comparison of access methods for time-evolving data," *ACM Computing Surveys*, vol. 31, pp. 158–221, June 1999.

[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, pp. 86–124, Feb 1989.

[7] 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.

[8] A. Fiat and H. Kaplan, "Making data structures confluently persistent," in *Proceedings of the 12th annual ACM-SIAM symposium on Discrete algorithms*, pp. 537–546, 2001.

[9] D. Sleator and R. E. Tarjan, "Self-adjusting binary search trees," *Journal of the ACM (JACM)*, vol. 32, no. 3, pp. 652–686, 1985.

[10] P. F. Dietz, "Maintaining order in a linked list," in *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pp. 122–127, 1982.

[11] S. Alstrup, T. Husfeldt, and T. Rauhe, "Marked ancestor problems," in *Proceedings of the 39th Annual Symposium on Foundations of Computer Science*, p. 534, 1998.

[12] M. Odersky, P. Altherr, V. Cremet, I. Dragos, G. Dubochet, B. Emir, S. McDirmid, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, L. Spoon, and M. Zenger, *An Overview of the Scala Programming Language*, Technical Report LAMP-REPORT-2006-001. 2nd ed., 2006. Online: http://www.scala-lang.org/docu/files/ScalaOverview.pdf.

[13] A. Allombert, G. Assayag, M. Desainte-Catherine, and C. Rueda, "Concurrent constraints models for interactive scores," *Proceedings of Sound and Music Computing Conferences (SMC)*, pp. 14:1–14:8, 2006.

[14] U. A. Acar, G. E. Blelloch, and K. Tangwongsan, "Non-oblivious retroactive data structures," tech. rep., CMU-CS-07-169, Carnegie Mellon University, School of Computer Science, 2007.