

# Proposal for an XML format representing Time, Positions and Parts of Audio Waveforms

**Jens GULDEN**

Formal Models, Logic and Programming (FLP),  
Technical University Berlin  
jgulden@cs.tu-berlin.de

**Hanns Holger RUTZ**

Studio for electroacoustic Music (SeaM)  
HfM Weimar  
hanns.rutz@hfm-weimar.de

## Abstract

A domain-specific model describing the notions of *time*, *positions* and *parts* of audio waveforms and other media is proposed. The model is primarily intended to be part of an overall XML specification for musical instruments and other sound generators, but appears also applicable to other time-based media. A practical software-implementation would provide functionality to mark points and regions in audio waveforms, apply annotational metadata to these positions and parts, cut an audio waveform into pieces and recombine them in a different order, or extract parts of a waveform as individual content.

A prototype implementation that realizes a functional subset is available.

## Keywords

Audio, Time, Slice, Position, Metadata

## 1 Introduction

We describe a proposal for modeling the notions of *time*, *positions* and *parts* of audio waveforms. The model is also applicable for other kinds of musical pieces and time-based media.

Section 2 motivates the need for a common interchangeable standard for locating positions in audio waveforms and marking parts of them. The following section 3 then sketches a conceptual model from scratch which provides the basis for implementing an XML data format. In section 4 such an initial XML-based implementation of the model gets introduced. Section 5 shows a prototypical implementation using the XML format, written in SuperCollider 3. The current status of sound-slices support as implemented by the real-world audio file editor Eisenkraut is covered by section 6. Finally, section 7 sketches future plans and section 8 gives a brief concluding summary.

The appendix contains an XML-Schema definition which formalizes the proposed XML language.

## 2 Need for a common file format for slicing audio waveforms

The value of multimedia data highly increases with the ability to structure it and refer to individual parts and positions of time-based media rather than treating every physical media file as one unstructured monolithic block. Every additional dimension of structuring broadens the variety of potential uses (e. g. attaching metadata to time-based media), and also unfolds a wide range of possibilities for artistic transformations.

However, until today even for audio data no standard of interoperability for marking positions and specifying regions is available. The field of possible applications for such a standard is huge: it ranges from creative musical activities in transforming audio, e. g. creating sound-collages or break-beat loops, audio-engineering applications in mixing / mastering, to possibly any private usage when attaching annotations to an audio file, e. g. let your friend read “this is the best guitar-solo I ever heard” right in the moment when the corresponding part is played from a music file.

When composing with sound, or creating realtime interactive pieces and installations, the proper sectioning of audio files and the tagging of time positions becomes vital. Therefore, new application fields for the classical sound-editor tool emerge, which is demonstrated later in this article.

## 3 Proposed Conceptual Model

### 3.1 Requirements

While terms like “time” and “position” seem clearly defined in a physical sense (locating coordinates in 4D time-space), in a musical context “time” and “position” have more facets and multiple dimensions of meaning. The notions of times and

positions in musical pieces are rarely used in an absolute sense. Instead, they usually depend on each other and are also relative to musical characteristics as e. g. the tempo of a piece, or a waveform's sample-rate. It is thus vital to an adequate domain-specific model of times and positions in musical pieces not to be restricted to a flat, absolute view on times and positions (as for example the number of milliseconds or sample-frames from the start), but to treat times and positions as recursively interdependent. The model proposed here captures these notions.

Further conceptual requirements that arise from this domain-specific interpretation of the terms are:

- *Time* in a musical piece may simply refer to clock time (e. g. '5.2 s'). But especially in the context of music, time may also refer to a beat-measurement scale (e. g. 'three...and'-beat) or, when handling waveform data, may be given as number of sample-frames. Time can also be understood as being relative to a part (e. g. 'two-thirds of part A').
- *Positions* in musical pieces are understood as inherently relative to other positions (e. g. 'at beat 3 after the end of the second verse'), not just as absolute positions denoted by single time-value. Thus, a *position* is specified recursively by referring to another *position*, combined with a *time* offset that denotes the distance between the two positions.
- *Parts* of musical pieces can be related to each other in several different ways, but *parts* may also be completely unrelated to each other. The specification is thus required to allow flexible declarations of *parts*, e. g. giving parts in a consecutive sequence, placing them in a hierarchy (as *parts* of other *parts* (, ... of other *parts*, ... etc.)), or let them freely overlap.

### 3.2 Terminology

To embed the above requirements into a formalized model, conceptual terms and additional helper terms are introduced as model elements. These basic model-terms are as follows:

#### “Waveform”

The physical representation of a sound or musical piece. A waveform has a *sample-rate* and a *length in frames* associated with it, sometimes a tempo can be specified as a *beats per minute* measure (which may default to a fix value and remain unused on sounds that are not tempo-structured), and an *offset* can be specified which denotes the actual start of an

audio-piece in the waveform. Optionally, a waveform can also have a *name* to refer to it. Every waveform owns an implicit reference to exactly one *master-slice*, from which any number of *slices* can be derived.

#### “Anchor”

A technical alias for "position in a waveform". This is sometimes called “marker” synonymously. *Anchor*s are always specified relative to another *anchor*, often the *start-anchor* of a parent-slice in the waveform (or implicitly the beginning of the entire waveform, which expresses an absolute position). As each anchor refers to another anchor to which it is relative to, the concept of anchors is inherently a recursive one, and so is the concept of *slices*:

#### “Slice”

A technical alias for "part of a waveform". A *slice* is delimited by a start-anchor and an end-anchor. Every slice also references exactly one *parent-slice*. The positions specified by the start-anchor and end-anchor usually are relative to the parent-slice. In most cases, slices also have a name which identifies them uniquely.

A slice is recursively specified as always having a parent-slice associated with it. This way, a slice is always a child of another slice, which again is a child of another slice, etc.

Different slices may overlap.

The top-most slice which represents a whole waveform is called the *master-slice*.<sup>1</sup> The master-slice never gets created explicitly, instead there is an implicit 1:1-relationship between each waveform and one master-slice representing it. (I. e., every waveform 'automatically' has one master-slice associated with it.) See the specification of the <wave>-element below in section 4 XML implementation of the model.)

#### “Time”

Time in a sound or musical piece. As demanded by the requirements, *time* can be specified either in number of beats, in sample-frames, as time in seconds or milliseconds, or relative to the length of the current parent-slice. Such different possible descriptions of *time* are named *time-values*.

---

<sup>1</sup>The idea of a master-slice is introduced as a helper-concept to mark a starting point of the recursive slices-hierarchy. It is not vital to the overall model and might get replaced in future versions of the model, e. g. by allowing any slice to take the role of a top-most slice.

The time offset between a relative base-*anchor* and a specified *anchor* is also represented by an instance of *time*.

“Time-Value”

A symbolic string-value from which *time* can be derived. The string is composed of a numeric part and an optional suffix which determines its type (beats, sample-frames, seconds, etc.). The suffix can be

- "~", for beat measurements ([measr:]beat.frac)
- "#", for sample-frames.
- "s", for seconds
- "ms", for milliseconds
- or none for a relative value between 0.0 (start of the parent-slice) and 1.0 (end of the parent-slice).

Note that the same *time* can be expressed by different *time-values*.

“CueList”

One possible way to gain practical use from cutting musical pieces into slices is to build a new sequence of musical pieces from the slices. As an example This way, e.g., a song could first be divided into slices of verses, choruses, bridge-parts etc., and later get combined in a different order. Such a re-combined sequence is called a *cueList*.

A *cueList* is usually composed of multiple *cues*.

“Cue”

A reference to a *slice*, to be used inside a *cueList*. The advantage of combining *cues* to *cueLists* (and not directly combining slices to possible 'slicelists') is that by using *cues* as explicit references to *slices*, confusion about identity and multiplicity is avoided. Multiple occurrences of the same *slice* inside a *cueList* become cleanly modeled as multiple distinct *cues* that reference one and the same *slice*. For convenience, each *cue* should be repeatable / loopable *n* times, so that multiple repetitions of *slice-occurrences* can be described by a single *cue*.

Visualization and Overview

A UML class-diagram ([3]) showing the model concepts and their relationships is displayed in Fig. 1. Table 1 summarizes the terms introduced with the model.

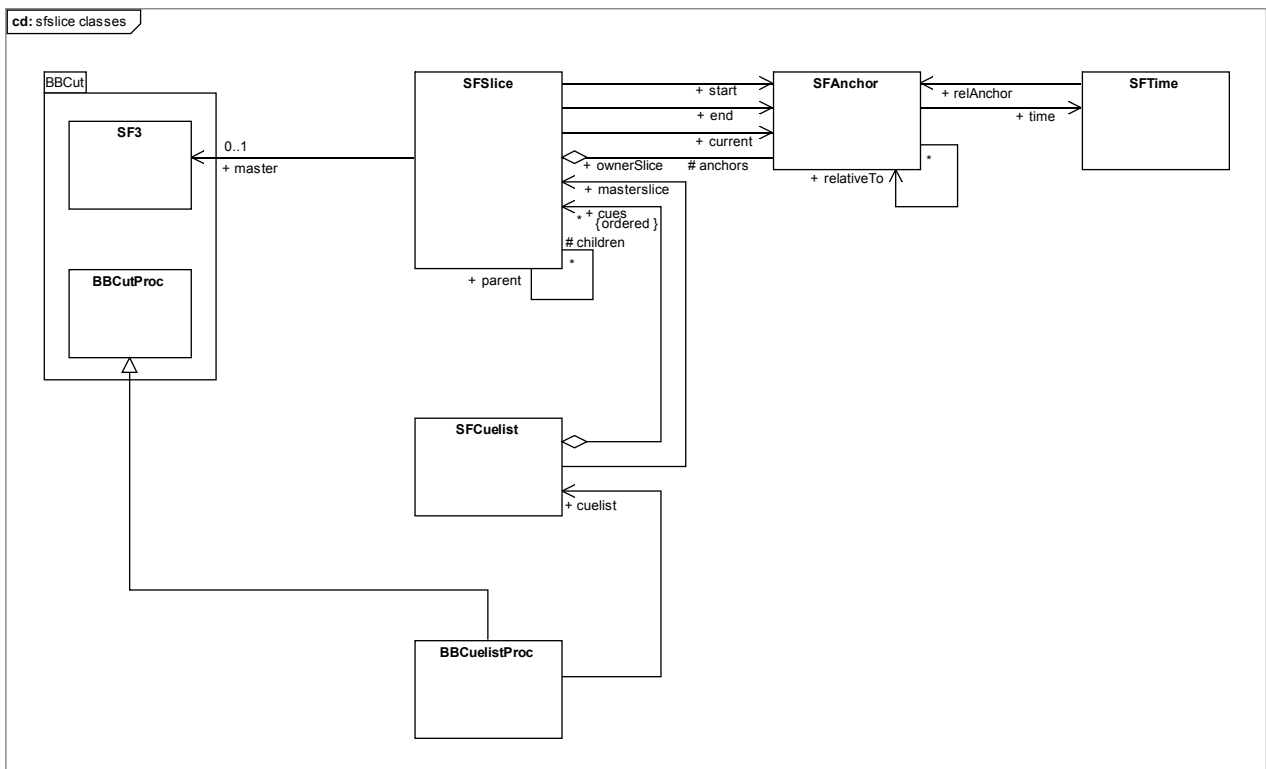


Fig. 1: UML class-diagram of the model

Term	Description
<i>Waveform</i>	The physical representation of a sound or musical piece. Every waveform owns an implicit reference to exactly one master-slice.
<i>Slice</i>	Part of a waveform. A slice is delimited by a start-anchor and an end-anchor.
<i>Anchor</i>	Position in a slice. Always specified relative to another anchor.
<i>Time</i>	Time in a sound or musical piece, either specified as number of beats, in sample-frames, as time in seconds, or relative to the length of the parent-slice.
<i>Time-Value</i>	Description of time. Different time-values can denote identical points in time (because there are multiple different ways to specify time).
<i>Cue</i>	A reference to a slice.
<i>Cuelist</i>	Combined sequence of cues.

Table 1: Summary of introduced terms

## 4 XML implementation of the model

The conceptual model developed in the previous section is now used as a basis for specifying a set of XML-elements which make up an XML language implementation for structuring audio data.

### 4.1 Overall Example

Example 1 gives an introductory overview of how a waveform is sliced into parts and re-joined as a cuelist in which the original parts are played in a different order and with repetitions.

```

<instrument>
  <wave src="examples/audio/NosNod0rr.wav">
    <!-- wavefile is implicitly used as 'master-slice' -->
    <slice name="start" to="20.55839s"/>
    <!-- empty slice, starts at end of previous slice: -->
    <pad to="27.69878s"/>
    <!-- next slice, starts at end of previous slice: -->
    <slice name="zwischen" to="34.83637s"/>
    <!-- relative time-values: -->
    <slice name="wummer" from="0.33" to="0.5"/>
    <!-- full syntax variant: -->
    <slice name="basis">
      <start>
        <anchor time="34.9s"/> <!-- shortcut-anchor -->
      </start>
      <end>
        <anchor> <!-- full anchor syntax -->
          <time>
            <seconds>49.1638</seconds>
          </time>
          <!-- <time value="49.1638s"/> (shortcut) -->
        </anchor>
      </end>
    </slice>
    <!-- specifying length instead of end-anchor: -->
    <slice name="funny" from="49.19s" length="7.14s"/>
    <!-- length only, starts at end of previous slice: -->
    <slice name="melody1" length="10.26s"/>
    <!-- time-value via frame-position: -->
    <slice name="melody2" to="42256#"/>
    <!-- relative time-value: -->
    <slice name="end" to="1.0"/>
  </wave>
  <!-- a cuelist for playing slices in a different order -->
  <cuelist name="song">
    <cue slice-ref="start"/>
    <cue slice-ref="zwischen"/>
    <cue slice-ref="wummer" repeat="4"/>
    <cue slice-ref="basis"/>
    <cue slice-ref="funny"/>
    <cue slice-ref="melody1" repeat="2"/>
    <cue slice-ref="melody2"/>
    <cue slice-ref="basis"/>
    <cue slice-ref="melody1" repeat="2"/>
    <cue slice-ref="melody3"/>
    <cue slice-ref="wummer" repeat="8"/>
    <cue slice-ref="funny"/>
    <cue slice-ref="melody1"/>
    <cue slice-ref="melody2"/>
    <cue slice-ref="melody3"/>
    <cue slice-ref="end"/>
    <cue slice-ref="melody2"/>
    <cue slice-ref="melody1"/>
    <cue slice-ref="melody2"/>
    <cue slice-ref="melody3"/>
    <cue slice-ref="end"/>
  </cuelist>
</instrument>

```

Example 1: sliced waveform in XML representation

Let's look at this example step by step:

The `<instrument>`-element represents an outer container that initiates the document. In an overall XML specification for musical instruments, `<instrument>` could carry far more functionality than here.

The `<wave src="..">`-element declares a waveform to be loaded.

The `<slice>`-elements inside the `<wave>`-element describe parts into which the waveform is divided. Note that the conceptual model requires all `<slice>`-elements to appear inside other slices (their parent-slice) or to explicitly reference a parent-slice. The `<wave>`-element implicitly provides a master-slice, so `<slice>`-elements can as well appear inside `<wave>`-elements to use the master-slice as their parent-slice.

## 4.2 Shortcut syntax vs. full syntax

One general aim of the language design is to both support manual editing of the XML document and allow easy machine writability / readability. This is why several declarations are supposed to be expressible either by the use of a shortcut syntax variant for manual editing, or by a full-size syntax in which every model concept is represented by its own XML-element. Generally, the shortcut-syntax allows to specify some model-concepts as attributes, while the full-size syntax makes use of one separate XML-element per model-concept.<sup>2</sup>

An implementing application may decide to only support either of the syntax variants, e. g. only the full-sized syntax if the XML format is intended to be used as a file-format for data-storage only (serialization / deserialization of objects from an object-oriented software-environment is easy to achieve in full-size syntax).

For means of readability, most of the upcoming examples make use of the shortcut-syntax.

## 4.3 Specifying slices

### *Shortcut syntax*

These are examples of how the time-values of start-anchors and end-anchors of a slice could be specified via XML-attributes:

```
<slice name="test" from="0.1s" to="3.5s"/>
<slice name="test2" from="2.5~" to="1.0"/>
```

Example 2: specifying time-values via attributes

---

<sup>2</sup>The distinction between a shortcut-syntax and a full-sized syntax is uncommon to XML-language design, as XML usually is used as an internal data storage format only and human readability is not a design-goal. We believe, however, that in the domain of artistic work on musical material, it is desirable to provide a manually editable view on the model which can undergo any artistic transformation by hand.

Using this shortcut form, anchors are declared by specifying time-values via the attributes `from`, `to` and `length` (e. g. `<slice from="1.5s" to="3.75s"/>`). Time-values of different types can be distinguished via a suffix in the string, which could be `~` for beat measurements, `#` for sample-frames, `s` for seconds, `ms` for milliseconds, or none for a relative value between 0.0 (start of the parent-slice) and 1.0 (end of the parent-slice). See also section 3.2 “Time-Value”.

Different ways of using the `from` and `to` attributes in `<slice>`-elements allow to distinguish between different semantics of the shortcut notation: If both a `from` and a `to` attribute are given, the slice should be placed between two anchors which are implicitly created at the specified times.<sup>3</sup> If only a `from` attribute is given, a start-anchor should implicitly get created at the time given by the `from` attribute, and the end of the slice should be denoted by the end-anchor of its parent-slice. Finally, if only a `to` attribute is given, the slice should start immediately behind the previously declared slice (in terms of XML, its previous sibling), or at the beginning of the parent-slice, if the current `<slice>`-element is the first one in its hierarchy level.<sup>4</sup>

```
<wave name="mywave" src="/data/wav/loop3.wav">
  <slice name="intro" to="12.305s"/>
    <!-- first slice starts at time-value 0.0 -->
  <slice name="vocals1" to="37.975s"/>
    <!-- identical: from="12.305s" to="37.975s" -->
  <slice name="bridge1" to="49.534s"/>
</wave>
```

Example 3: context-dependent defaults for `from` and `to` attributes

---

<sup>3</sup>It should be a convention to understand start-times as being included in the slice, describing the first discrete point in time that lies within the slice. On the contrary, end-times should be understood as being excluded from the slice, denoting the first discrete point in time after the end of the slice. (Discrete points in time could e. g. be counted by the sample-frame index.)

<sup>4</sup>This introduces context dependency in the parsing-process which is more complex than stand-alone XML-parsers are designed to handle by themselves. However, keeping track of such a context is relatively easy if parsing is done by traversing the document-tree on an application-level ('manually' programmed), or when serializing / deserializing objects to or from XML which usually allows the execution of class-specific code while parsing.

A slice's overall position could thus result from specifying either of the following options:

1. Independently specifying references to the start and the end anchor:

```
<slice from="time-value" to="time-value"
name="..."/>
```

2. Setting a start anchor and letting the end-anchor follow after a given time duration:

```
<slice from="time-value" length="time-value"
name="..."/>
```

3. Setting an end-anchor and letting the start-anchor appear earlier by a specified time duration:

```
<slice to="time-value" length="time-value"
name="..."/>
```

4. Setting an end-anchor and using the start-anchor of the most previously declared slice (or the parent's start-anchor if the declared slice is the first one on its hierarchy level):

```
<slice to="time-value" name="..."/>
```

5. Using the end-time of the most previously declared slice as a start-anchor's time and creating an end-anchor of the specified duration after the start-anchor:

```
<slice length="time-value" name="..."/>
```

6. Or using the long form for explicit anchor creation inside `<slice>`-elements, see the following sub-section.

A parser should be aware of these different combinations and report an error if invalid attribute sets (e. g. all three `from`, `to` and `length`-attributes set at the same time) are encountered.

### Full-size syntax

Slices, anchors and times can also be notated explicitly in a full-size XML notation variant. Using this syntax, the conceptual relations between these entities, as they have been expressed by the conceptual domain model, can get reflected directly by the corresponding XML elements `<slice>`, `<anchor>` and `<time>`.

Just as slices can be notated either in a detailed long-form or a shortcut-form, anchors may also be declared either by the use of a full-size syntax, or alternatively by making use of a shortcut `time`-attribute inside the `<anchor>`-element.

```
<!-- full-size slice and anchor declarations: -->
<slice name="myslice">
  <start>
    <anchor name="my_unique_start_anchor">
      <!-- a name is given for also referring to this
            anchor from elsewhere -->
      <time>
        <beats>2.5</beats>
      </time>
    </anchor>
  </start>
  <end>
    <anchor ref="my_unique_end_anchor-declared_elsewhere"/>
  </end>
</slice>

<!-- full-size anchor declaration outside a slice: -->
<anchor name="my_anchor">
  <time>
    <beats>2.5</beats>
  </time>
</anchor>

<!-- shortcut anchor declaration: -->
<anchor name="my_anchor2" time="2.5"/>
```

Example 4: full-size and shortcut anchor declarations

### Referencing a parent-slice

In order to provide an easy way of referencing a parent-slice, it should be possible to implicitly specify slices as being children of another slice by placing their declarations inside a sub-tree of the parent's `<slice>`-element.

An alternative way of referencing a parent-slice without nesting a `<slice>`-element inside another `<slice>`-element (or a `<wave>`-element), should be provided by explicitly referencing the parent-slice via the `parent-ref` attribute.

```
<!-- implicitly reference a parent-slice through nesting: -->
<slice name="vocals1" from="9.35s" to="37.975s">
  <slice name="first-half-of-vocals1" from="0.0" to="0.5"/>
</slice>

<!-- explicitly reference a parent-slice by name: -->
<slice name="vocals1" from="9.35s" to="37.975s">
  <!-- ... -->
</slice>

<!-- not an XML-child of vocals1, but uses it as parent: -->
<slice parent-ref="vocals1" name="first-half-of-vocals1"
from="0.0" to="0.5"/>
```

Example 5: referencing parent-slices

### Referencing anchors

Just like slices can be referenced via identifier-names, it should also be possible to reference anchors by name. This is especially useful when multiple slices are to share the same anchors.

Additionally, the declaration of named anchors outside any slice should be possible, in order to make anchors referable from other elements.

```
<wave name="mywave" src="/data/wav/loop3.wav"/>
<slice name="vocals1" parent-ref="mywave">
  <start>
    <anchor name="my_unique_start_anchor">
      <!-- name for referencing from elsewhere-->
      <time>
        <beats>2.5</beats>
      </time>
    </anchor>
  </start>
</end>
  <anchor ref="my_unique_end_anchor"/>
  <!-- anchor declared elsewhere -->
</end>
</slice>

<!-- shortcut notation (the application is responsible
for distinguishing time-values from anchor-refs): -->
<slice parent-ref="mywave" name="first-part-with-vocals1"
from="0.0" to="my_unique_end_anchor"/>

<!-- a named anchor declared outside a slice: -->
<anchor name="global_start">
  <time>
    <seconds>2.35</seconds>
  </time>
</anchor>

<!-- multiple references to the same named anchor: -->
<slice name="all" from="global_start" to="273.8s"/>
<slice name="vocals1" from="global_start" to="52.4s"/>
```

Example 6: named anchors

An implementation should evaluate references to declared elements after having completely parsed the XML document, in order to allow forward-references to identifiers.<sup>5</sup>

#### 4.4 Padding unused parts

In order to mark areas of the waveform as unused (among the group of children in one parent-slice), a `<pad>`-element should be provided.<sup>6</sup> A `<pad>`-element only makes sense in a sequence of `<slice>`-elements which are specified as consecutively following each other (i. e. tags of the form `<slice to="..." ... >` or `<slice length="..." ... >`). The `<pad>`-element could be used just like the `<slice>`-element, in order to declare an unused part of the sound or musical piece:

<sup>5</sup>This suggestion again imposes an additional requirement on context-sensitivity of the parser.

<sup>6</sup>From an implementer's perspective the `<pad>`-element might seem redundant, as it can equivalently be replaced by a dummy-slice which never gets used. However, we believe that from a domain-specific perspective additional semantics is expressed by explicitly marking a part as unused, which is why the `<pad>`-element is included in the proposal.

1. either by padding to a specific anchor (any slice or other pad created afterwards will start at that end time):

```
<pad to="time-value"/>
```

2. or by padding a specific duration of time:

```
<pad length="time-value"/>
```

#### 4.5 Specifying cuelists

A *cue* is a sequential collection of references to slices. Such an individual reference should be called a *cue*. Cues should allow slice-references to appear in any order and for any number of times in a cue. When the cue is played-back by an implementing application, it should sequentially output the slices in the same way as if they had been joined to be one major waveform.

It should be possible to attach a name to a cue via a name-attribute, in order to have a reference handle to the cue from a hosting application.

##### *Recursive use of cuelists*

Recursive declarations of cuelists inside cuelists should be possible. This is especially useful when the inner cue is to be repeated multiple times by use of the repeat-attribute.

A ref-attribute should allow to refer to cuelists declared elsewhere in the document.

```
<!-- a cue containing multiple cues: -->
<cue name="myCue">
  <cue slice-ref="intro" repeat="2"/>
  <cue slice-ref="vocals1"/>
  <cue slice-ref="bridge1"/>
  <cue slice-ref="intro"/>
  <cue slice-ref="vocals2"/>
  <cue slice-ref="chorus1" repeat="3"/>
</cue>

<!-- cue lists inside another cue: -->
<cue name="song">
  <cue slice-ref="intro" repeat="2"/>
  <cue slice-ref="vocals1"/>

  <cue repeat="2">
    <cue slice-ref="bridge1"/>
    <cue slice-ref="intro"/>
  </cue>

  <cue ref="more"/>
</cue>

<cue name="more">
  <!-- ... -->
</cue>
```

Example 7: some cue lists

## 5 Prototype implementation for testing

There is an experimental prototype software application that demonstrates some features of the proposed XML format. It is called the 'SFSlice classes' and written in the SuperCollider 3 [4] language, running on top of the BBCut [5] class library.

The application comes with an example audio file which gets sliced into several cues and is then re-combined to a new and longer musical piece as described by the supplied cuefile. Example 8 shows the supplied XML file which gets processed by the prototype to play a new song out of the sliced audio file.

The prototype exclusively uses the shortcut syntax.

```
<instrument>
  <wave src="examples/audio/NosMod0rr.wav">
    <slice name="start" to="20.55839s"/>
    <slice name="leer" to="27.69878s"/>
    <slice name="zwischen" to="34.83637s"/>
    <slice name="wummer" from="33.06s" to="34.82s"/>
    <slice name="basis" from="34.9s" to="49.16338s"/>
    <slice name="funny" from="49.19s" to="56.33s"/>
    <slice name="melody1" from="56.35s" to="63.46s"/>
    <slice name="melody2" from="63.5s" to="70.65s"/>
    <slice name="melody3" from="70.7s" to="77.8371s"/>
    <slice name="end" to="1.0"/>
  </wave>
  <cuefile name="song">
    <cue slice-ref="start"/>
    <cue slice-ref="zwischen"/>
    <cue slice-ref="wummer" repeat="4"/>
    <cue slice-ref="basis"/>
    <cue slice-ref="funny"/>
    <cue slice-ref="melody1" repeat="2"/>
    <cue slice-ref="melody2"/>
    <cue slice-ref="basis"/>
    <cue slice-ref="melody1" repeat="2"/>
    <cue slice-ref="melody3"/>
    <cue slice-ref="wummer" repeat="8"/>
    <cue slice-ref="funny"/>
    <cue slice-ref="melody1"/>
    <cue slice-ref="melody2"/>
    <cue slice-ref="melody3"/>
    <cue slice-ref="basis"/>
    <cue slice-ref="funny"/>
    <cue slice-ref="melody1"/>
    <cue slice-ref="melody2"/>
    <cue slice-ref="melody3"/>
    <cue slice-ref="end"/>
  </cuefile>
</instrument>
```

Example 8: XML example file running with the prototype implementation

The prototype implementation, along with example audio files and additional documentation of the conceptual model and the XML implementation, is available for download at [1].

## 6 Real-world application Eisenkraut

Finally we shall look at a concrete application which handles waveforms and time positions. Eisenkraut is a cross-platform audio file editor that utilizes the SuperCollider 3 server for realtime audio processing [2].

In the current beta-version 0.7, only anchors (called "Markers" here) are supported, but the implementation of slices (or "Regions") is planned. Markers are displayed in the traditional way as a separate track along with the waveform, where a linear timeline is forming the horizontal axis. In the underlying model, each track is associated with a list of regions which is double-sorted (by start frame and stop frame). This way editing operations such as cutting, pasting, moving etc. can be applied uniformly to any data that can be described by a start and stop frame.

While the underlying framework uses a sample frame timebase, on the user presentation level time-values can be specified in sample frames, milliseconds, hh:mm:ss or percentage (referring to the "implicit master slice" of the whole document span), as shown in Fig. 2:

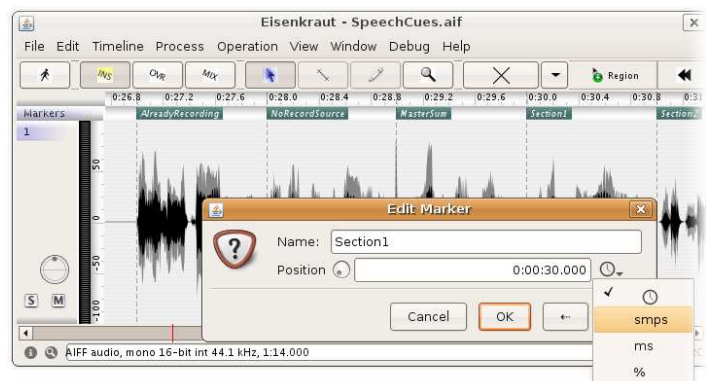


Fig. 2: editing a marker in Eisenkraut

Unfortunately, the notion of markers and regions varies depending on the audio file format used. For example, AIFF files [6] provide built-in support for markers but not regions. WAVE files [7] provide various concurrent concepts of both markers and regions.

While Eisenkraut currently uses the marker/region facilities of the particular audio file formats, the implementation of an independent time positions file as proposed in this paper is highly desirable for several reasons:

- The editing of marker positions is often a separate process independent of cutting and transforming the audio waveform. In existing audio file formats such as AIFF or WAVE, the deletion or creation of just a single marker

requires the whole audio file – often hundreds of megabytes – to be rewritten when saving

- Markers in audio files may not be meaningful to some stages of the processing. for example, in the SuperCollider 3 environment, streaming a soundfile off disk that contains hundreds or thousands of markers puts an unnecessary computation burden on the server objects (*scsynth*) reading the file, although the markers are only useful for client side objects (*sclang*).
- Markers can take different roles and often need to be associated with enhanced meta-data. For example, an audio file may be segmented in order to allow dynamic selections of material in a realtime performance or sound installation, requiring additional data such as fade-in and fade-out times, relative gain adjustments, descriptive attributes such as character-categories etc. Common audio file formats do not provide sufficient means to add these meta-data to markers.

The solution to this last issue is to copy the markers into custom (text or binary) files, separate from the audio files. Eisenkraut already facilitates this approach through the supply of an Open Sound Control (OSC) [9] server. Clients, typically written in SuperCollider, can interact with documents of the audio file editor in two ways:

- explicit OSC addresses and commands are provided to access the documents and its tracks. The following example shows how all markers of the current active audio file document can be retrieved from a SuperCollider client:

```
e = Eisenkraut.default;
e.addr.connect;

// queries all markers of the active document
(
fork {
  var msg, rate, num;
  rate = e.query( '/doc/active/timeline', \rate ).first;
  num = e.query( '/doc/active/markers', \count ).first;
  msg = e.get( '/doc/active/markers', [ \range, 0, num ] );
  msg.pairsDo({ arg pos, name;
    ("Marker "++name++" at frame "++pos++" = "
    ++(pos/rate).asTimeString( 0.001 )).postln;
  });
})
```

Example 9: reading markers through OSC

- inversely, the client can create new markers:

```
// create a geometric series of 20 markers across the file
(
fork {
  var len, scale, pos, names, marks;
  len = e.query( '/doc/active/timeline', \length ).first;
  pos = Array.geom( 20, 0.1, 1.2 );
  scale = len / ( pos.last * 1.2 );
  pos = pos.collect({ arg t; (t * scale).asInteger });
  names = Array.fill( 20, { arg i; "Mark #" ++ (i+1) });
  marks = (pos ++ names).unlace( 20 ).flatten;
  e.listSendMsg( [ '/doc/active/markers', \add ] ++ marks );
})
```

Example 10: adding markers through OSC

- implicit OSC addresses and commands are provided through SwingOSC [9]. This framework allows the clients to create and control custom GUI elements inside the Eisenkraut application. A large set of specialized GUI classes for SuperCollider is provided. Consequently, the editor itself can be kept small and generic, while custom marker handling and manipulation dialogs can be programmatically added by the client, as depicted in Fig. 3:

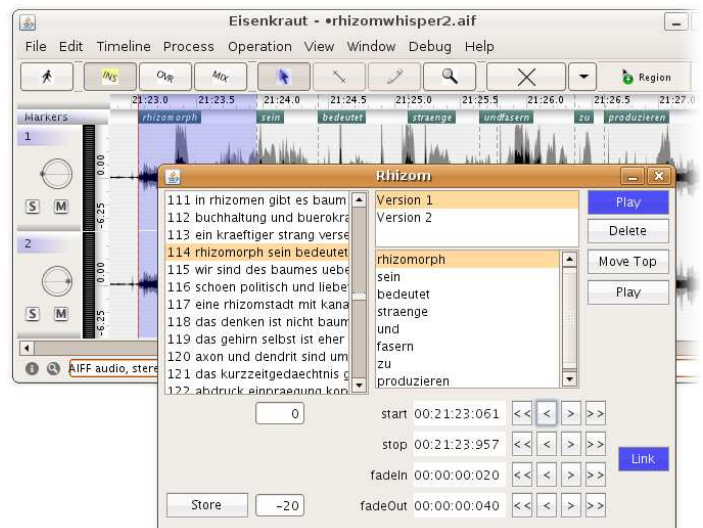


Fig. 3: custom editor using SwingOSC

- although Eisenkraut does not yet provide any container hierarchies for markers, the client can well present them in containers as indicated by the three list views. Also “meta-data” like fade-in and fade-out times as well as relative gains per region are presented in the client view. The client takes responsibility for storing the markers, in fact using XML files in this case.

Some additional work needs to be carried out to smoothen the interaction between OSC client and Eisenkraut. For example, markers could be tagged with flags, indicating their colour or indicating whether they should be saved along with the default

audio file or separately by the client. This way, the greatest flexibility can be maintained without making presumptions about the nature of the markers in the host application.

With this architecture, a new role is given to the traditional tool of the “soundfile editor”. Instead of taking the road of merging soundfile editors with multitrack recording applications – as for example exercised in Audacity [10] –, the audio file editor becomes a core *modular* element inside a *networked* environment for the composition of electroacoustic and noise music, realtime performance and sound installation. It is in all forms of musique concrète that the structuring process departs from the analysis and editing of sonic materials, chunking of time being the most fundamental element in this structuring.

A final example is presented in Fig. 4. The image shows a symbolic score for a sound installation in the former prison of the GDR's Ministry of State Security (MfS) in Erfurt, Germany, in summer 2006 [11]. A kind of timeline goes from top to bottom, traversing five different stages or four sound materials. Each sound material is shown as a circular shape, and markers are used for segmentation of each material (indicated by spokes). Again this is a simplification as fading characteristics are not displayed. As offsets and lengths of the sections are chosen randomly, one particular performance is indicated by blue sectors.

All markers have been transferred from Eisenkraut to SuperCollider using drag-and-drop (as the OSC server did not yet exist at that time). Also note that the region structure is partly linked together, as indicated by the connections between the materials “Wald Stein” and “Wald Atmo”. In this case, a binary file format was chosen for both the flat and the hierarchical markers, however the presence of a standardized XML format and supportive libraries could have simplified the process.

## 7 Future Plans

The given proposal is an initial approach which has already started a discussion on improvements and alternatives. It will develop over time, and as the data format might get implemented in other applications, modifications will flow back from there.

Additional potential for further development lies in broadening the focus from audio-data to any time-based media, such as video, or even non-time based media like written documents. In these cases, slices could denote scenes, cuts or chapters and

paragraphs. Time-values could be extended to be also specifyable as video-frames, i-frames, page-number / line-number combinations etc.

## 8 Conclusion

The proposed model covers several issues that have been identified as specific to the domain of marking positions and regions in audio waveform data. It has been shown that these tasks are of practical importance, as confirmed by experiences with the development of the Eisenkraut audio file editor.

The model and its XML implementation propose a number of profoundly elaborated constructs to fulfil musically relevant requirements. The approach thus can be used as an initial basis for further elaborating the model and the XML format. It is one contribution to a much wider range of interoperable data formats describing musical events and other time-based media which yet wait to be invented.

## References

- [1] Gulden, J., *Proposal for an XML Specification modeling Time, Positions and Parts of Audio Waveforms*, conceptual model and software, <http://swiki.hfbk-hamburg.de:8888/MusicTechnology/786>
- [2] Rutz, H. H., *Eisenkraut – cross-platform audio file editor*, software, <http://www.sciss.de/eisenkraut>
- [3] Gulden, J., *Model-Driven Software-Development with SuperCollider and UML*, this conference
- [4] McCartney, J., *SuperCollider – A real time audio synthesis programming language*, software, <http://supercollider.sf.net>
- [5] Collins, N., *BBCut SuperCollider 3 extension library*, software, <http://www.cus.cam.ac.uk/~nc272/bbcut2.html>
- [6] *Audio Interchange File Format* <http://www.borg.com/~jglatt/tech/aiff.htm>
- [7] *WAVE File Format* <http://www.borg.com/~jglatt/tech/wave.htm>
- [8] Open Sound Control, <http://www.opensoundcontrol.org>
- [9] Rutz, H. H., *SwingOSC – Open Sound Control server to script the java language*, software, <http://www.sciss.de/swingOSC>
- [10] Dominic Mazzoni et al., *Audacity – The Free, Cross-Platform Sound Editor*, software, <http://audacity.sourceforge.net>
- [11] Rutz, H. H., *untitled (Zelle 148)*, installation, <http://www.einschluss.de>
- [12] Sun Microsystems, *Sun Multi-Schema XML Validator*, software, <https://msv.dev.java.net>

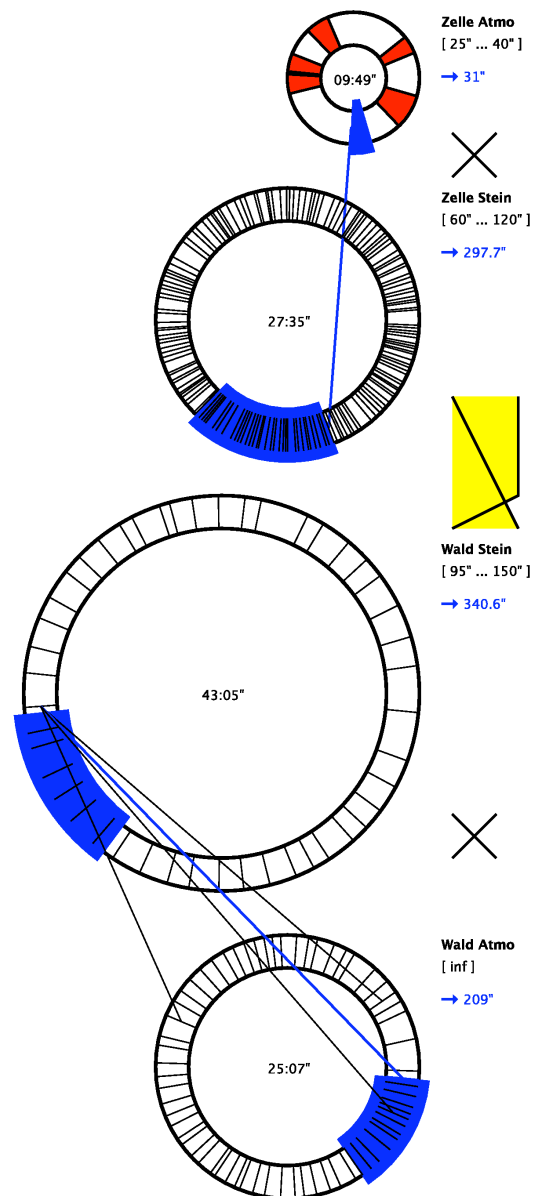


Fig. 4: Score of 'untitled (Zelle 148)', indicating audio file segmentations

## Appendix: XML-Schema Definition (XSD)

```

<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  name="instrument">
  <xs:element name="instrument">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="wave"/>
        <xs:element ref="cuelist"/>
        <xs:element ref="slice"/>
        <xs:element ref="anchor"/>
      </xs:choice>
      <xs:attribute name="name" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="wave">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="slice"/>
        <xs:element ref="pad"/>
      </xs:choice>
      <xs:attribute name="name" type="xs:string"/>
      <xs:attribute name="src" type="xs:string"
        use="required"/>
      <xs:attribute name="tempo" type="xs:decimal"/>
      <xs:attribute name="offset" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="slice">
    <xs:complexType name="slice-type">
      <xs:all minOccurs="0" maxOccurs="1">
        <xs:element ref="start"/>
        <xs:element ref="end"/>
      </xs:all>
      <xs:all minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="slice"/>
      </xs:all>
      <xs:attribute name="name" type="xs:string"/>
      <xs:attribute name="parent-ref" type="xs:string"/>
      <xs:attribute name="from" type="xs:string"/>
      <xs:attribute name="to" type="xs:string"/>
      <xs:attribute name="length" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="pad">
    <xs:complexType>
      <xs:complexContent>
        <xs:restriction base="slice-type">
          <xs:all minOccurs="0" maxOccurs="1">
            <xs:element ref="start"/>
            <xs:element ref="end"/>
          </xs:all>
          <xs:attribute name="name"
            type="xs:string"
            use="prohibited"/>
          <xs:attribute name="from"
            type="xs:string"/>
          <xs:attribute name="to"
            type="xs:string"/>
          <xs:attribute name="length"
            type="xs:string"/>
        </xs:restriction>
      </xs:complexContent>
    </xs:complexType>
  </xs:element>
  <xs:element name="anchor">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="1">
        <xs:element ref="time"/>
      </xs:choice>
      <xs:attribute name="name" type="xs:string"/>
      <xs:attribute name="ref" type="xs:string"/>
      <xs:attribute name="time" type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="start">
    <xs:complexType>
      <xs:sequence minOccurs="1" maxOccurs="1">

```

```

        <xs:element ref="anchor"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="end">
    <xs:complexType>
      <xs:sequence minOccurs="1" maxOccurs="1">
        <xs:element ref="anchor"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="time">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="1">
        <xs:element name="beats" type="xs:decimal"/>
        <xs:element name="frames" type="xs:integer"/>
        <xs:element name="seconds" type="xs:decimal"/>
        <xs:element name="milliseconds"
          type="xs:decimal"/>
        <xs:element name="relative" type="xs:string"/>
      </xs:choice>
      <xs:attribute name="value"
        type="xs:string"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="cuelist">
    <xs:complexType>
      <xs:choice minOccurs="0" maxOccurs="unbounded">
        <xs:element ref="cue"/>
        <xs:element ref="cuelist"/>
      </xs:choice>
      <xs:attribute name="name" type="xs:string"/>
      <xs:attribute name="ref" type="xs:string"/>
      <xs:attribute name="repeat" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
  <xs:element name="cue">
    <xs:complexType>
      <xs:attribute name="slice-ref" type="xs:string"
        use="required"/>
      <xs:attribute name="repeat" type="xs:integer"/>
    </xs:complexType>
  </xs:element>
</xs:schema>

```

All XML-examples presented in this article have been validated against this schema using the Sun Multi-Schema XML Validator ([12]).