CHRONICLE: REPRESENTATION OF COMPLEX TIME STRUCTURES

Wijnand Schepens
University College Ghent, Belgium
wijnand.schepens@hogent.be

ABSTRACT

Chronicle is a novel open source system for representing structured data involving time, such as music.

It offers an XML-based file format, object models for internal representation in various programming languages, and software libraries and tools for reading and writing XML and for data transformations.

Chronicle defines basic blocks for representing time-based information using events, a hierarchy of groups and instantiable templates. It supports two modes of timing: local timing within a group and association with other elements. The built-in mechanism for resolving time references can be used to implement both timescale mappings and tagging of information.

Chronicle aims to be a powerful and flexible foundation on which new file formats and software can be built. Chronicle focuses on structure and timing, but leaves the actual content free to choose. Thus format- or software-developers can specify their own domain-model. This makes it possible to make representations for different types of musical information (scores, performance data, ...) in different styles or cultures (CMN, non-western, contemporary, ...), but also for other domains like choreography, scheduling, task management, and so on. It is also ideal for structured tagging of audio and multimedia (movie subtitles, karaoke, synchronisation, ...) and for representing "internal" data used in music algorithms.

The system is organized in four levels of increasing complexity. Software developed for a specific level and domain will also accept lower level data, while users can choose to represent data in a higher level and use Chronicle tools to reduce the level.

1. INTRODUCTION

Since the beginning of computing hundreds of music encodings (representations, formats) have been invented, and new ones are still being developed today. For overviews see e.g. [1–3]. Part of the reason is that the landscape of forms of musical information is so large, ranging from audio to symbolic, from performance to score, from western to non-western, from classical to popular, from ancient to contemporary. Some of these terrains are relatively well established, even standardized, others are still being explored.

Most symbolic music encodings focus on common western music notation (CWMN). However, many new applications are pushing the limits of conventional encodings. Some examples:

- non common western music, e.g. traditional African music, medieval music
- special notations such as percussion notations, tablature, Gregorian plainchant
- extra data such as lyrics, choreography, instrument-specific notations, harmony. Also: auxiliary data for music software, e.g. chroma vectors, onset times, ...
- synchronization or alignment with multimedia, annotations and labeling

Existing encodings are not always suitable to accommodate the storage of these types of data. Developers are forced to invent their own encoding, or to abuse existing encodings such as MIDI [1].

The most important common factors in these applications are time and structure. We have developed a new system for dealing with structured data involving time, called *Chronicle*. The system deals with time and structure, but leaves the actual content or data free to choose. A developer has the freedom to represent the content in any form he wishes. Thus the Chronicle system can serve as the skeleton for a variety of applications, allowing the developer to concentrate on content-specific issues.

In the early 90s, one of the first systems dealing with time was HyTime [4] on which the symbolic music encoding SMDL [5] was based. Although both are international ISO/IEC standards, and have had a lot of influence on later initiatives, they have never been used in practice. There are a number of reasons for this, but the main reason is probably that the system was far too complex. Chronicle is similar in some aspects, but aims to be as simple as possible. Related efforts for symbolic music can be seen in Music Space [6] and SMI [7] or IEEE 1599 [8].

Similar time-based approaches can be found in the field of multimedia, notably with SMIL [9] and its recent offspring Timesheets [10]. For a more theoretical background please refer to [11]. These systems were designed to schedule multimedia presentations. The use of parallel and sequential groups of elements (sound, image, movie) can also be found in Chronicle, but Chronicle offers much more advanced mechanisms for annotations, timescale mappings, templates etc.

Chronicle is intended to aid developers of (music) software and file formats, by providing simple yet powerful and flexible basic building blocks and structuring mechanisms. It offers support for working with events, groups, relative time, parallel and sequential layout, timescales, timescale mappings, association, parametrized templates and more. The system was developed primarily for symbolic music, but is also applicable in other domains like audio, multimedia, choreography, job scheduling etc.

The Chronicle system consists of different parts:

- external representation: XML-based file format
- software tools for manipulating Chronicle XML files
- internal representations: object models (interfaces, classes, ...)
- software libraries for manipulating, storing, parsing...

Internal representations and libraries are developed in various programming languages [1] .

The main aim is to facilitate the development of new domain-specific encodings and software by providing internal data representations (in the form of interfaces and classes) and software-libraries for various tasks such as writing and parsing XML, processing events, manipulating structure, querying information etc.

2. DESIGN

2.1 Elements and ID's

The basic elements are *events* and *groups*. A group contains child-elements, which are either events or sub-groups. Sub-groups contain other elements and so on. The root-group is the common ancestor of all elements. The result is a hierarchy or tree-structure, comparable to a file system. Every element in the tree, except the root, has a *parent group*.

Every element is identified within its parent group by a unique integer number called *local* ID. By default elements are numbered 0, 1, 2, ..., but it is possible to override this by an explicit ID-definition.

Every element in the hierarchy is uniquely identified by its local ID, the local ID of its parent group, the local ID of its grandparent group etc. This sequence of local IDs is called the elements *global* ID.

It is convenient to introduce *path-notation*, where a global ID is specified by concatenating the local IDs top-down (starting from the root) separated by slashes, similar to a file path or URL. The root itself is notated by `/`. Thus, for example, global ID `/2/3` identifies the element with local ID 3 in the sub-group with local ID 2 in the root group. Optionally, an element can have a name (String) which is also unique within its parent group. Names can be used to embellish path-notation, e.g. `/part1/section3/4`.

[1] currently Java and ActionScript

Path expressions starting with `/` are called *absolute*, because they identify an element starting from the root. It is also possible to identify elements using a *relative* ID from within another element. As in a file-path, one can use ".." in path-notation to indicate the parent (group). For example `../../section/2` refers to the element with local ID 2 in a group named "section" in the grandparent group of the current element. As an alternative, the notation `@`*`name`* can be used to refer to an element named *name* anywhere up in the hierarchy. First the parent group is searched. If the element is not found there, the grandparent is searched, and so on, until the root is reached. This mechanism is similar to the lookup of variable names in nested scopes. For example, `@section/2` is equivalent to `../../section/2` if the grandparent contains a child named "section", but the parent doesn't.

2.2 Time

All elements have a timestamp, which is expressed either in *local time* or in *non-local time*.

Local time is represented by an integer number and is to be interpreted relative to the timescale of the parent group. The parent group has its own (start-)time and, optionally, a scale. An example in XML-form:

```
<group time="100" scale="4">
   <event time="10" />
</group>
```

In this fragment the "absolute" time of the event would be $100+4\times10 = 140$. If an element's time is not specified, it defaults to zero (local time). The scale defaults to one.

Alternatively, an element can specify its timestamp in non-local time using a *time reference*. A time reference is represented by a path-expression, either absolute or relative, which can be written using path-notation starting with `/`, `..` or `@`. If this path refers to an element which exists in the tree, then the referring element gets the same time as the element it refers to.

```
<event name="e1" time="100" />
...
<group>
  <event name="e2" time="@e1" />
</group>
```

In this example event "e2" refers to event "e1", so it also has time 100. Here we used `@e1` in the path expression. Equivalently, one could specify this using a relative path `../e1` or an absolute path.

Time references can also be chained (refer to a reference...) and mixed with local timing:

```
<group name="e1" time="100" />
  <event time="10" />
  <event time="20" />
</group>
...
<group time="@e1/0">
  <event name="e2" time="1" />
</group>
```

Here event "e2" uses local time, which is added to the parent group time, resulting in `@e1/1`. This path refers to the second element (id=1) of the group ”e1”. The result is that the time of event ”e2” resolves to $100 + 20 = 120$.

The technique of time references is very powerful. It can be used for associating elements with other elements (tagging...), but also to implement mappings between timescales. This will be illustrated in section 3.

Chronicle also supports layout-schemes which allows automatic determination of element times. The most common examples are sequential and parallel layout which schedule elements resp. one after the other and at the same time.

2.3 Levels

Chronicle is organized in four levels of increasing complexity.

A level 0 file consists of one list of events. There is one global timescale. All events use local time relative to this timescale. The events are ordered in ascending time order: every event must have a time greater than or equal to that of its predecessor. No restrictions are imposed on the data carried by the events.

Level 1 adds the possibility of groups and nesting, with the restriction that groups have starting time equal to zero. This means that local times in groups are equivalent to ”absolute” times in the global timescale. In this (and higher) levels, the events needn’t be ordered in time.

In level 2 all elements, including groups, can specify their (start) time as a local time or using a time reference, and groups can use a layout-scheme to set child element times.

Finally, level 3 introduces templates and template instances. Note that every level is a subset of the higher levels. Chronicle provides tools to transform data to a lower level.

2.4 Domain

As noted in the introduction the Chronicle system only deals with structure and timing, not with the actual domain-specific content, which can be chosen freely. Different Chronicle *applications* can be developed for specific domains representing different types of musical information (scores, performance data, ...) in different styles or cultures (CMN, non-western, contemporary, ...), but also for other domains like choreography, scheduling, task management, and so on. It is also ideal for structured tagging of audio and multimedia (movie subtitles, karaoke, synchronisation of score and audio, ...) and for representing ”internal” data used in music algorithms (chroma, coordinates, ...)

In the XML-format domain-specific information is encoded *in the content of event-elements*, either as text or as one or more child elements. Additionally it is possible to give the event an attribute `type`. To illustrate this we show some possibilities for encoding a chord-symbol:

```
<event>Am7</event>
<event type="chord">Am7</event>
<event><chord>Am7</chord></event>
<event>
  <chord><root>A</root><kind>m7</kind></chord>
</event>
```

We have provided API’s to read and write Chronicle elements (event, group, ...), but it is up to the user to deal with event-content. Groups cannot carry data themselves, although it is possible to specify a group-type using the `type`-attribute. If necessary, extra events can be introduced in a group to encode group-related information.

3. EXAMPLE

We will demonstrate the key features of the Chronicle system and the level reduction process by means of an example. We present the material in XML-form, although it should be borne in mind that Chronicle also supports ”internal” representations and transformations in different programming languages.

```
<chronicle version="2.0" level="3"
  domain="http://.../leadsheet" />

<template name="note" >
  <group>
    <event time="0">
      <note_on  pitch="#pitch" />
    </event>
    <event time="#dur">
      <note_off pitch="#pitch" >
    </event>
  </group>
</template>

<template name="voice">
  <group>
    <group name="notes" layout="sequential">
      <instance model="/note"
                pitch="D4" dur="12" />
      <instance model="/note"
                pitch="D4" dur="12" />
      <instance model="/note"
                pitch="D4" dur="9" />
      <instance model="/note"
                pitch="E4" dur="3" />
      <instance model="/note"
                pitch="F#4" dur="12" />
      ... MORE NOTES ...
    </group>
    <group type="lyrics" time="@notes/0">
      <event time="0" type="lyric">Row,</event>
      <event time="1" type="lyric">row,</event>
      <event time="2" type="lyric">row</event>
      <event time="3" type="lyric">your</event>
      <event time="4" type="lyric">boat</event>
      ... MORE LYRICS ...
    </group>
  </group>
</template>

<group name="song" time="@ticks/0">
  <group name="canon" layout="sequential">
    <group time="0" name="first">
      <instance model="/voice" />
    </group>
    <group time="48" name="second">
      <instance model="/voice" />
    </group>
    <group time="96" name="third">
      <instance model="/voice" />
    </group>
    <group time="144" name="fourth">
      <instance model="/voice" >
    </group>
  </group>
  <instance name="unison" model="/voice">
</group>

<group name="ticks" time="@ms/0" >
```

```
  <event id="0"   time="0" />
  <event id="192" time="19200" />
  <event id="240" time="21600" />
</group>

</chronicle>
```

This example illustrates an encoding of the song ”Row, row, row your boat”. It is important to note that this is only one of many possible encodings.

After the XML-preamble, the file starts with a root-element `chronicle`. The attribute `version` specifies the version of the Chronicle system itself. The attribute `level` indicates the encoding level used, and the attribute `domain` specifies the domain (content types and structural restrictions). In this case the attribute points to a URI.

The first element defines a template `note`. A template is a kind of prototype which can be instantiated (copied) multiple times. Templates are useful for avoiding code-duplication. A template can have parameters, which makes it possible to vary the instances. In this case the `note`-template is used as a convenient way to bundle a note-on and note-off event. It represents a single note with a certain pitch and duration. The pitch is encoded as a simple string which indicates pitch class (e.g. `F#`) and register or octave (4th octave). This is not dictated by Chronicle but is a choice made by the domain-developer.

Next, the template `voice` defines notes and lyrics of the song. The group `song` instantiates five copies at different times, representing four voices sung in canon, followed by one in unison. Finally, group `ticks` relates the ticks-timescale to milliseconds.

The example is a level-3 encoding. We will now illustrate how it can be reduced to lower levels.

The first phase, which transforms from level-3 to level-2, is template instantiation, also called expansion. It is carried out bottom-up: first the innermost elements, then their parents and so on. In this case, the voice-template contains instances of note-templates which are instantiated expanded first, the parameters `#pitch` and `#dur` being substituted by their actual values defined in attributes `pitch` and `dur`. Subsequently, the four voice-instances in the song-group are expanded. Since this template has no parameter, the instances are exact copies. In the resulting level-2 file the templates have disappeared:

```
<chronicle version="2.0" level="2"
  domain="http://.../leadsheet" />

<group name="song" time="@ticks/0">
  <group time="0" name="first">
    ...
  </group>
  <group time="48" name="second">
    <group>
      <group name="notes" layout="sequential">
        <group>
          <event time="0">
            <note_on  pitch="D4" />
          </event>
          <event time="12">
            <note_off pitch="D4" />
          </event>
        </group>
        ... MORE NOTES ...
        <group>
          <event time="0">
            <note_on  pitch="E4" />
          </event>
          <event time="3">
            <note_off pitch="E4" />
          </event>
        </group>
        ... MORE NOTES ...
      </group>
      <group type="lyrics" time="@notes/0">
        <event time="0" type="lyric">Row,</event>
        <event time="1" type="lyric">row,</event>
        <event time="2" type="lyric">row</event>
        <event time="3" type="lyric">your</event>
        <event time="4" type="lyric">boat</event>
        ... MORE LYRICS ...
      </group>
    </group>
  </group>
  ... THIRD AND FOURTH VOICE ...
</group>

<group name="ticks" time="@ms/0" >
  <event id="0"   time="0" />
  <event id="192" time="19200" />
  <event id="240" time="21600" />
</group>

</chronicle>
```

Reduction from level-2 to level-1 is carried out in two phases. The first phase is the *layout* phase. In the example, the `notes`-group has sequential layout, which means that its elements must be scheduled one after the other. Technically, the (start) time of element $i + 1$ is equal to the (start) time of element i plus the duration of element i, for all i.

The duration of a group is equal to the largest local time of (grand)child events. If necessary the duration can also be specified explicitly. In the case of the notes, the duration of a note-group is equal to the time of the offset-event.

In the resulting file the element groups have acquired an explicit time, and the layout-indications are gone. In the example, the note-groups have times 0, $0 + 12 = 12$, $12 + 12 = 24$, $24 + 9 = 33$, $33 + 3 = 36$ and so on. This is illustrated in the following fragment which shows the onset-event of the fourth note (E4) in the second voice, and its ancestor groups:

```
<group name="song" time="@ticks/0">
  <group time="48">
    <group>
      <group name="notes">
        ...
        <group time="33">
          <event time="0">
            <note_on  pitch="E4" />
          </event>
```

The layout phase is followed by the *time resolution* phase. The timestamps of all elements are resolved in the manner illustrated in section 2.2.

Consider for example the onset of the fourth note (E4) in the fragment above. Following the way up from parent to parent, it can be seen that the additions result in a time equal to `@ticks/81`. This means that that note starts on tick 81.

The domain-developer has chosen to encode the lyrics in a separate group which is *associated* with the notes-group. In the group lyrics the event times are added to the group-time, yielding value @notes/0, @notes/1 and so on. The @notes-reference points to the notes-group. Therefore the timestamps of the lyric-events are substituted by the timestamps of the note-groups which they refer to, in this case @ticks/0, @ticks/12 and so on.

The reference @ticks in turn points to the ticks-group defined near the end of the example. As this group contains only three events with local ID 0, 192 and 240, a reference like /ticks/12 doesn't point to a real element. Such a reference is called *virtual*. In that case times are resolved by a linear interpolation between real elements. In the example ID 0 maps to time 0 and ID 192 maps to 19200, so each tick in this range has a duration of 100 ms. This means, for instance, that @ticks/12 resolves to @ms/1200 which signifies that tick 12 occurs after 1200 milliseconds. Ticks in the range up to 240 have a duration equal to $(21600-19200)/(240-192) = 50$ ms. As a result, the fifth instance of the voice-template (named "unison") is played in double tempo.

The net effect is that the ticks-group defines a mapping between timescales. Note that the mechanism for resolving (local or non-local) times is used to accomplish two different goals: a. the lyrics are associated (tagged) to notes, b. the ticks timescale is mapped to the millesecond timescale.

Note that all references can be resolved to a form ms/x where x is an integer number. The reference @ms/... cannot be resolved any further - this is the "global" timescale. If we set all group times to zero, then the absolute times are equivalent to relative times. The result is a level-1 file, with the timescale ms specified in the root-element:

```
<chronicle version="2.0" level="1"
  domain="http://.../leadsheet"
  timescale="ms" />

<group name="song">
  <group name="first">
    ...
  </group>
  <group name="second">
    <group >
      <group name="notes" >
        <group>
          <event time="4800">
            <note_on  pitch="D4" />
          </event>
          <event  time="6000">
            <note_off pitch="D4" />
          </event>
        </group>
...
        ... MORE NOTES ...
        <group>
          <event time="8100">
            <note_on  pitch="E4" />
          </event>
          <event time="8400">
            <note_off pitch="E4" />
          </event>
        </group>
...
        ... MORE NOTES ...
      </group>
      <group type="lyrics" >
        <event time="4800" type="lyric">
Row, </event>
        <event time="6000" type="lyric">
row, </event>
        <event time="7200" type="lyric">
row </event>
        <event time="8100" type="lyric">
your </event>
        <event time="8400" type="lyric">
boat </event>
...
        ... MORE LYRICS ...
      </group>
    </group>
  </group>
  ...
  ... THIRD AND FOURTH VOICE ...
</group>
...
... TICKS ...
</chronicle>
```

To reduce from level-1 to level-0 one more transformation is needed. In this final phase groups are *serialized* or *flattened* into one long series of events, and they are ordered in ascending time order. In the resulting level-0 file, there are no more groups:

```
<chronicle version="2.0" level="0"
  domain="http://.../leadsheet"
  timescale="ms" />
...
<event time="4800">
  <note_on  pitch="D4" />
</event>
<event time="4800" type="lyric">Row,</event>
<event  time="6000">
  <note_off pitch="D4" />
</event>
...
<event time="8100">
  <note_on  pitch="E4" />
</event>
<event time="8100" type="lyric">your</event>
<event time="8400">
  <note_off pitch="E4" />
</event>
...
</chronicle>
```

4. IN PRACTICE

How can the Chronicle system be used in practice?

The XML-format can be used by software-developers as a convenient means to persist data. Chronicle provides easy-to-use libraries for writing and reading the XML-format. For new software projects it may even be advisable to use the Chronicle classes and interfaces as the basis for the object model, even if XML-persistence is not needed. It is up to the developer to specify the domain-model by constraining content types and level.

Consider, for example, a Chronicle-encoding of MIDI-information (see e.g. [1]) By its very nature, this application is ideal for a level-0 encoding, i.e. a flat list of simple events. The domain model establishes event-types (note-on, note-off, control change, program change) and their XML-encoding. A typical event could look like this:

```
<event time="196">
  <note_on key="100" velocity="127" channel="0">
</event>
```

Software applications operating on this data, for example for playing the music, typically process the events one by one. Whereas the processing software is happy to consume level-0 data, from a musical point of view it may be desirable to add some structure. This can be achieved by encoding in a higher level, using mechanisms such as grouping, association, sequences, templates etc. Chronicle tools can then be used to transform to level-0 and feed the reduced data to the processing software.

The nice thing is that users can choose the organization which best suits their needs. For example, one might choose a "part-by-part" organization using parallel groups for different instrument parts, or a "frame-by-frame" organization using a sequence of parallel note-groups. One can choose to use sequential groups for measures, for sections (movements). It is also possible to play around with associations, timescales and time-mappings, and so on.

One important point which hasn't been addressed is that Chronicle files can be embedded as subgroups within another chronicle file using an `include`-element. The mechanism of association by time-references can be used to add information to a group without changing it. This is especially useful if the target is an embedded group, or if it is external data such as an audio or multimedia file.

Chronicle is currently being tested for the encoding of leadsheets for *wikifonia.org*. In the near future, we are planning to create more domain-specific applications, and hope that other developers will do the same.

Documentation, tools and open source code can be found at **http://code.google.com/p/chronicle-xml/**.

5. REFERENCES

[1] E. Selfridge-Field: *Beyond MIDI: The Handbook of Musical Codes*, MIT Press, Cambridge MA, 1997.

[2] K. Ng, and P. Nesi: *Interactive Multimedia Music Technologies*, Information Science Publishing, 2007.

[3] R. Cover: "XML and Music," retrieved July 6, 2009 from http://xml.coverpages.org/xmlMusic.html, 2006

[4] C. Goldfarb: Standards: "HyTime: A standard for structured hypermedia interchange," *IEEE Computer magazine*, 24(8), 1991.

[5] D. Sloan: "Aspects of Music Representation in HyTime SMDL.," *Computer Music Journal*, 17, 51-59, 1993.

[6] J. Steyn: "Introducing Music Space," *Proceedings of the 4th Open Workshop of MUSICNETWORK: Integration of Music in Multimedia Applications*, Barcelona, 2004.

[7] G. Haus, and M. Longari: "A multi-layered, time-based music description approach based on XML," *Computer Music Journal*, 29, 70-85, 2005.

[8] L. Lucidovo: IEEE 1599: "a Multi-layer Approach to Music Description," *Journal of Multimedia*, 4(1), 2009.

[9] D. Bulterman, J. Jansen, et al.: "Synchronized Multimedia Integration Language (SMIL 3.0)," retrieved July 6, 2009, from http://www.w3.org/TR/2008/REC-SMIL3-20081201/, 2008.

[10] P. Vuorimaa, D. Bulterman, and P. Cesar: "SMIL Timesheets 1.0 - W3C Working Draft," retrieved July 6, 2009, from http://www.w3.org/TR/2008/WD-timesheets-20080110/, 2008.

[11] S. Boll, U. Klas, and W. Westermann: "A Comparison of Multimedia Document Models Concerning Advanced Requirements," *Technical Report Ulmer Informatik-Berichte* No 99-01, 1999.