Implementing the SMIL Specification

Malcolm Tredinnick
CommSecure
malcolm@commsecure.com.au

Abstract

Synchronized Multimedia Integration Language (SMIL) is a W3C recommendation for encoding multimedia presentations. It provides presentational control over not just the spatial layout of the document, but also the relationship between elements over time. At the present time there does not appear to be a high quality Open Source implementation of the SMIL 2.0 specification available. This paper describes one attempt at an implementation. Some ideas about where future software development could take this implementation to fulfill the requirements of other projects are also mentioned.

1 A Potted History Of SMIL

1.1 SMIL 1.0 — June 1998

In mid-1998, the W3C promoted the SMIL 1.0 specification [4] to Recommendation status. The goals of this recommendation were (from its abstract)

1. to describe the temporal behavior of a presentation,
2. describe the layout of the presentation on a screen, and
3. associate hyperlinks with media objects.

Here, media objects are either continuous media, such as video or audio presentations, or discrete media such as text blocks and images.

The SMIL 1.0 specification was not overly complex. It was not short (about 30 pages when printed out), but that was because it introduced a number of elements with specific semantics that needed to be explained clearly. Implementations appeared, but SMIL did not set the world on fire. This was possibly because although it fulfilled the design requirements mentioned above, that was *all* it did. A presentation needed to be separated out into individual media elements in practice and leveraging existing content was difficult[1].

1.2 SMIL 2.0 — August 2001

Roughly three years later the SMIL 2.0 specification reached Recommendation status [5]. Superficially, this specification looked vastly different from the SMIL 1.0 version. It was also much larger (the 1.0 document was around 150 KB in size, 2.0 was 3.3 MB). However, upon closer inspection, it became apparent that the changes fell broadly into a only a few categories.

1. Additional markup for transitions between media components, extra layout

[1] This is just the present author's observation. The real reason may be that SMIL 1.0 was ahead of its time; a solution in search of a problem.

possibilities, content control (so that presentations on PDAs and large monitors could be driven from the same source) and extra events for controlling the presentation.

2. Improved accessibility features (for example, offering a choice between closed captioning and audio descriptions).

3. A modular design so that SMIL modules could be reused in conjunction with other XML modules. This also provides a way to allow small-footprint implementations to implement the commonly used portions of the language and omit the more complex parts whilst still being able to process documents intended for more featureful implementations.

SMIL 2.0 provided backwards-compatibility with SMIL 1.0 documents and precisely laid out how to process older documents with respect to the newer features. Content written for the older specification did not become obsolete. In view of this, an implementation of "SMIL" need only focus on the 2.0 specification since 1.0 compatibility comes for free[2].

Today, SMIL is *still* not setting the world on fire. It is slowly gaining visibility, though, and is becoming incorporated into a number of auxiliary specifications which is forcing it into peoples' consciousness. We consider a few of these uses later in this paper.

2 Building A SMIL Library

Around June of 2002, the present author stumbled across SMIL whilst looking for something entirely different and started to read the specification, as one does in those circumstances. In November, a rough design for a library to parse SMIL documents and somehow present them was sketched out. Over the following weekends and the odd evening or two, code started to come together to the point where at the present time[3] a complete implementation of the specification is within sight. The implementation has been done without reference to third-party products, since the whole idea was that it was meant to be a fun project and a test of how difficult it might be to implement from the specification alone.

Existing SMIL implementations all appear to be designed along the lines of building a complete presentation application for SMIL documents. The library being considered in this paper, `libsmil`, has different goals.

The `libsmil` library parses a SMIL document and extracts the data into a format that can easily be used by a client application to make the presentation. It is then up to the client application to start and stop the presentation of the various media objects in windows as directed by the library (see Figure 1).

In essence, `libsmil` manages a bunch of data structures containing information about the presentation for the client. From time to time, the library will poke the client and tell it to start or stop presenting on of the media objects in a particular location. Similarly, the client will call the library whenever the user (or other external stimulus) triggers an event that may influence the progress of the presentation. This encompasses actions like clicking on a "stop" button or following a hyperlink to some other location in the document. The library itself is agnostic about the means used to present the information by the client. This should provide the opportunity to reuse this library in a variety of different situations, since it has very few dependencies on its own and

[2] Well, it comes after a lot of work, in practice. But you still only need to implement a single specification.

[3] As this paper is being written in April 2003

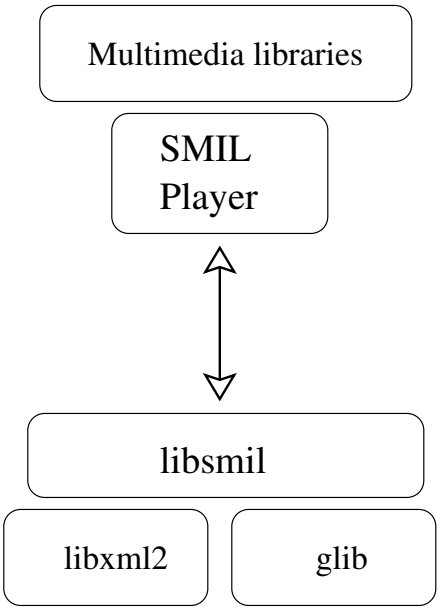

Figure 1: Basic `libsmil` architecture

places few restrictions on the client's behavior and operation.

3 An Interaction With A Presentation Client

In order to illustrate the features provided by `libsmil`, we consider a typical series of calls between the library and a client application. For our purposes here, it suffices to understand that a standalone SMIL document consists of a `head` and a `body` portion, much as XHTML does. The head of the document contains data about the presentation and the body contains the presentation itself. The explanation in this section necessarily skips over some of the minor steps, but we will endeavor to touch on all the highlights.

3.1 Pre-Processing

After initializing `libsmil`, the client reads in the document and creates a DOM representation using the GNOME XML library `libxml2`. This responsibility is given to the client since an internal representation of the data to be presented may already exist or is being created on the fly. This would be typical, for example, in the scenario where the client is using SMIL as part of a larger language, such as XHTML+SMIL (see [8]). For more information on this type of usage, see section 5.

The DOM is then passed to `libsmil` for initial, pre-presentation processing. Initially, this involves parsing all of the contents of the `head` element

The library extracts from the document the regions that are going to be used in the presentation and what their dimensions are. A SMIL document can have a number of top-level regions (essentially, separate windows for a GUI client) and each top-level region can contain any number of named regions within it. These internal regions can be used to display videos, captions, text overlays on images or videos or whatever the content author desire. The regions may overlap and can have a Z-ordering applied to them so that one region always appears on top of another region. Each region can also have attributes specified to indicate how media within it is treated in terms of clipping, scrolling or scaling.

A SMIL document can specify any number of customized tests which are used in the body of the document. These are a sequence of named data elements which may contain arbitrary values. The values can be set by the user—through some interface on the client—or via a URI. Some of the tests are marked as intended for user setting, whereas others are nominated as "hidden," meaning that although an advanced client may permit them to be set, they will typically take their value either from a default setting or by retrieval from some remote location. So `libsmil` extracts out all the custom tests, sets up their default values

and types (visible or hidden) and puts them into a data structure for the client to present to the user upon request.

In the final phase of the pre-presentation scan, the library makes a pass over the `body` element of the document and tries to extract two more pieces of data.

1. A list of all the media types that are required to make the presentation, and
2. a graph of all the timing and synchronization dependencies between the elements in the body of the document.

The first item here is straightforward: every media object is marked with one of a limited number of tags and includes a compulsory attribute indicating the type of the media. Building up a list of required media types and linking the types to the URIs for the contents is just a matter of parsing these elements. This list is given back to the client, who can use it to pre-load the appropriate presentation libraries or prepare itself (and the user) for the fact that some items will not be presentable due to the lack of an appropriate output method.

The timing and synchronization data is what makes implementing the SMIL specification interesting and challenging. It is non-trivial! Rather than interrupt the flow of this section, we will just take it as given that some magic happens and a timing graph is constructed which `libsmil` will use during the presentation to control events. In section 4 we will come back to this problem, since it lies at the heart of the benefits SMIL can provide to external applications. The timing graph that is constructed here is only of use to the library and will remain hidden from the client. That is a fair division of knowledge: most of the complexity of a presentation lies in getting the timing issues correct and that is ultimately what the client is relying on the library for.

After collecting all this data, control is returned back to the client. The client can now query the library to extract a list of top-level regions that it will need to provide, or to find out which media formats are going to be required. The client then initializes any extra libraries it requires to make the presentation and registers a single callback with `libsmil`.

3.2 Running The Presentation

After all the data collection in the previous section, running a presentation is a reasonably simple process, with a couple of exceptions mentioned below.

The client program calls `smil_run_presentation()` to start the action[4]. If the user does nothing, the majority of the presentation will consist of `libsmil` calling the function that the client had registered at the end of the last section. This function contains instructions to start or stop the presentation of a particular media element in a particular region. For continuous media, a "start" instruction may also contain an offset at which to begin playing the fragment, a repeat count and possibly a time multiplier. This last option is an advanced feature in SMIL that permits authors to alter the natural duration of a media fragment by, in effect, accelerating or decelerating time as seen by that item only. Not all players will be able to support this feature and presentation authors need to allow for that possibility.

From time to time, the user may trigger an event at the user-interface. This could be something like clicking on a hyperlink, hitting a hotkey, or closing a window on the desktop. The client will pass this event, along with any asso-

[4]This must be done in a thread or separate process, since at the same time as `libsmil` is creating events, the client must be responding to user interaction *and* presenting the media to the user.

ciated contextual information such as the name of the link or region that was clicked, back to the library. These events will be used by the library to control the future of the presentation, but from the client's point of view they simply result in further calls to the registered callback for starting and stopping presentations.

The only slightly complex feature when running a presentation is how to present features from the Animation modules. These SMIL modules provide a means to change the value of attributes on elements over time. The values can change linearly and non-linearly over time. They can jump to discrete values at different moments. The value against time graph can even be specified using Bezier splines.

The Animation modules are most commonly used when SMIL is incorporated into another language profile. So the attributes being changed might be things like the display style property on an XHTML element, or the length of a line in SVG. Displaying the host language elements correctly is clearly the job of the client. However, managing the complex value against time relationships is something that the SMIL library should be doing, since it possesses complete knowledge of the required algorithms.

Currently, when an `animation` element is begun, `libsmil` calls the client as normal with a start instruction and supplies the appropriate initial values for the attribute(s) in question. Subsequently, whenever the client is ready to redraw the element being animated, it calls `smil_update_animation(...)` with the given element / attribute pair identifier and retrieves the new value for the attribute that is being animated. This is a rare case of the client pulling the information it needs, rather than waiting to be notified of an update. Due to the small number of users of `libsmil` at the time of writing, it is unclear if this method of running an animation is the best. As more experienced is gathered, this interface may change.

The other exception to the normal run algorithm outlined above is precipitated by the Transition Effects module. This module provides a number of transitions that can be used when moving between media objects within the same region. They include various fades and screen wipes. A SMIL implementation (a player) supporting the Transition Effects module is required to support at least four transitions. However, the specification outlines over 100 transitions, with a fallback algorithm for when a transition is specified that the player cannot handle.

In the present implementation, a client is required to know the required behavior for each of the transitions it supports. For example, the library might call the client with an instruction to begin the horizontal windshieldWipe with a duration of three seconds. How the client implements this wipe is left up to the client. It is possible (but not compulsory) for the client to tell `libsmil` which wipes it can handle so that the library can implement the fallback algorithm on behalf of the client. This way of using the library is consistent with keeping as many SMIL-related decisions in the library's domain as possible.

A client can take this to extremes and tell the library that it can do no transitions at all—that is, it does not support the Transition Effects module—in which case `libsmil` will simply optimize away the calls to begin and end transitions. This would be appropriate, for example, in a client that is designed to present audio and braille information for blind users; implementing wipes makes no sense in that situation.

This implementation is not perfect—it is one case where the client is required to have knowledge of the SMIL specification in some de-

tail. The alternative, however, is for the library to pass back a bitmap of how a region should look as the wipe progresses. The problem with this is that it would involve putting knowledge about presentational techniques into a library that is otherwise devoid of such information. It would also remove some possibilities for the client to optimize or improve the wipe according to circumstances. For example, the same wipe performed on a region that is 300 by 200 pixels may look different on a PDA than on a screen capable of much higher resolutions and with more CPU power available.

In practice (limited as it may be), this implementation appears to work. The minimal four transitions required by the module are trivial to implement in a client. Some multimedia libraries, such as the `gstreamer` library from the GNOME project, also provide ways of doing standard wipes (the SMIL specification uses a number of wipes already specified in standards for the television and motion picture industries). Therefore, requiring clients to implement their own versions of each wipe may not be particularly onerous. Once again, continued use of `libsmil` should provide better indications on this front.

4 The Timing And Synchronization Module

As mentioned in section 3.1, the heart of `libsmil` is the timing and synchronization code. This is the longest and possibly the most complex portion of the specification. The `libsmil` implementation has been rewritten three times so far and although it is approaching completeness, testing continues in an effort to try and gain some confidence that the behavior is correct in all circumstances. A fourth (or further) rewrite is not out of the question as this code is required to be both easy (or even just *possible*) to maintain and fairly fast, since it is where most of the library's work is done while a presentation is running.

Without going into too many details, SMIL contains three container elements for holding content that is temporally related. The `seq` ("sequential") element contains items that are presented one after the other in the order they are given in the document. The `par` ("parallel") element contains items that are to be presented in parallel, subject to time constraints on their begin and end times and lengths. Finally, the `excl` ("exclusive") element wraps content of which only one item can be playing at any given time, although the order is not important. One might use `excl` to present a number of video clips from amongst which the user can select arbitrarily with each new clip replacing the one that is currently playing.

Within each of these containers, each element can have a begin time (absolutely specified or relative to the start of the container), an end time, a duration and a repeat count[5]. Further to this, the containing element (the `seq`, `par`, or `excl` element) can also have begin, end, duration, and repeat counts attached to it. After all, this container may reside inside another temporal container and so on. In fact, this last possibility is universally true. All elements are assigned a behavior that is equivalent to one of the containers. By default, all elements, including and, in particular, `body`, act as `seq` elements. So everything in the document is played in order from beginning to end with well-defined semantics as it concerns scheduling.

By and large, scheduling the beginning and ends of elements inside a `seq` or `excl` element is fairly straightforward. At least, these cases certainly fall out easily after the logic for

[5] There are some restrictions on these values depending on the type of containing container, but we will treat them as all roughly the same here; no real confusion should result from doing so.

```
<par>
   <video id="vid"
          begin="-5s"
          dur="10s"
          src="movie.mpg" />
   <audio begin="vid.begin+2s"
          dur="8s"
          src="sound.au" />
</par>
```

Figure 2: A sample `par` container holding video and audio elements.

```
<par>
  <img id="foo"
       begin="0; bar.begin+2s"
       dur="3s" .../>
  <img id="bar"
       begin="foo.begin+2s"
       dur="3s" .../>
</par>
```

Figure 3: A ping-pong effect.

handling the contents of a `par` is implemented.

The difficulty for a `par` lies in the fact that elements may have multiple begin times (and multiple end time, but we shall omit a detailed discussion of those here). These times may also be relative to the begin or end times of other elements, even those within the same `par` container. Further, an element's starting time may be before the starting time of its containing element. The contained element will not begin before its parent, so the net effect is that when it does begin, it will appear to have already been playing for some time.

An example may make this clearer (see Figure 2). In this example, when the containing `par` element begins, the video will begin playing five seconds into its length. Thus it will appear as though it started from the beginning five seconds *before* the `par` element started. The video will then play for five seconds, since its total duration is ten seconds and it effectively started at minus five seconds. The audio element in this container begins two seconds after the video begins. This is effectively at minus three seconds, from the point of view of the `par` container. Thus the audio will really start playing three seconds in from its beginning and will run for a further five seconds, ending at the same time as the video element. This is a very typical example of how audio and video sequences can be synchronized despite having possibly come from different sources.

In this example, we saw a case of an element that has a definite starting time (the `video` element) coupled with one that has an indefinite starting time (the `audio` element whose start time depends upon the `video` element). It is those elements with indefinite starting (and ending) times that can make life difficult for the implementation. In some cases, such as the above example, the effective start time of the element can be easily determined, even as early as the pre-presentation pass, since its only dependency is on something with a definite starting time. However, the start time could be dependent upon a something such as a button click event being sent, which is unresolvable until presentation time.

One significant test of any synchronization implementation is something like the code fragment in Figure 3.

The effect here is that the image labeled *foo* is displayed at time zero and lasts for three seconds, image *bar* is displayed initially two seconds into the element and lasts for three seconds. This triggers *foo* to be displayed again at four seconds into the element's period (the start time of *bar* plus two more seconds) and so on, back and forth between the two image. The duration of three seconds here is relatively meaningless: it could be anything longer than

two seconds and the same effect would occur.

Implementing the code to process this fragment requires a little planning. It appears that all of the indefinite time periods (the `bar.begin+2s` parts) can be resolved, since everything can be traced back to a dependency on the instance of *foo* that starts at time zero. However, there is no definite end to this element (although there may be one hidden in the omitted portion of the above example). It would obviously be a mistake to try and resolve all of the image start times out to infinity. Instead, `libsmil` detects that there is a loop in the dependency chain and stops resolving at that point. It then becomes a matter of resolving portions of the infinite dependency chain as required when the container element is being presented.

The examples that we have considered in this section are indicative of the problems to be solved by the timing and synchronization code. A glance at [5] shows that there are many more cases to consider, but the logic is fairly well explained in the specification. The problem is that there is just a lot of it and the interactions between cases is complex.

In theory, getting the timing information correct is just a problem in directed graph theory. In practice, it is a maze of twisty passages, all alike, and somewhat difficult to navigate correctly.

5 Using `libsmil` To Extend Other Languages

In section 1.2 it was explained that one of the changes between versions 1.0 and 2.0 of the SMIL specification was that modularity was introduced. This was done along the same lines as the XHTML modular design and for the same reasons—it enables the language to be extended or for portions to be lifted and dropped into another language profile in order to extend the latter. It is natural, therefore, to try and design `libsmil` in such a fashion that it can assist with presenting these extension languages.

On the whole, this has not been too difficult. The languages that one would choose to extend with SMIL are things like XHTML, SVG and MathML—languages which normally are static presentations once rendered. SMIL adds the ability to change attribute values over time, particularly via the Timing and Synchronization modules and the Animation modules.

Using `libsmil` to render a document in, say, SVG+SMIL is very similar to rendering a pure SMIL document. The library does a prepresentation pass over the document to build up information about the nodes it will influence and to create a time graph, just as in the standalone case. Once this pass is finished, the client renders the document using the initial values for all attributes. It then calls `smil_run_presentation()` and waits for the registered callback to be triggered with the usual instructions about starting or stopping some action. In this case, these actions will typically be things like changing the value of an attribute, rather than playing a media item.

The main difference from the standalone case that will arise is when the document being displayed is changed by some event outside the control of `libsmil`. In the standalone case, all document navigation is controlled by `libsmil`; in the extension case, the SMIL library does not have the knowledge of how navigation works in the external (hosting) language, so that is up to the client to manage. Therefore, the client may from time to time call `smil_new_document()` to load a completely different document or `smil_jump_to_xpath()` to move to a location

within the current document.

For client applications that already have decent access to their documents' parsed object model, adding support for SMIL's temporal activity appears not to be too difficult.

6 Applications Of SMIL

In case the reader is still wondering about the practical benefits of SMIL, which have probably not been made clear in the previous sections, here are a small number of typical use cases.

Recorded presentations It is possible to coordinate the automatic presentation of a conference speaker's slides with the audio recording of the their talk. The slides will progress at the right moment. Extra navigation possibilities for both the audio and visual portions of the talk can be presented as well.

Digital Talking Books SMIL is already part of the DAISY 2 [2] and ANSI/NISO Z39.86 [1] talking book standards—the latter standard being also known as DAISY 3. DAISY 2 requires SMIL 1.0 support, while Z39.86 requires a minimal SMIL 2.0 implementation. These two standards provide visually impaired people and people with reading difficulties a means to access literature that would otherwise be closed to them.

Captioning for video formats Many digital movie and video formats do not contain subtitles as sideband information. Sometimes, subtitles are provided, but not for the required language. The ability to synchronize a video presentation with arbitrary textual captions will provide a benefit both to people with hearing difficulties and to those watching a presentation given in a foreign language.

Educational presentations As authoring tools become available, pulling together disparate sources into a coherent presentation should become relatively straightforward. This will permit educators to begin to build up a library of coordinated presentations using information that currently might be scattered all over the Web. It was not mentioned earlier in this paper, but the media objects displayed by SMIL can be retrieved from remote URLs as well as local files. Also, SMIL provides a mechanism for pre-fetching any content that may take time to download so as not to hold up later portions of the presentation.

Kiosk and conference display front-ends SMIL provides a simple way to create a menu-based presentation. It can also revert to a standard looping presentation once the requested video or audio has completed. This makes it ideal for writing control documents for video kiosks or product displays at conventions.

7 Future Work

Development on `libsmil` is focused on creating a complete implementation of the specification. Simultaneously, some demonstration applications and a small presentation program are being written to show off the library's features.

Following on from that work, a number of obvious "next steps" present themselves. The following list is in no particular order, but all are achievable tasklets.

1. Implement the Timed Text specification that is currently being developed by the

W3C [7]. This will allow for scrolling captions and easier synchronization of captions with audio and video.

2. Implement a digital talking book player. Currently, no Open Source implementation of the DTB standards is available. With proprietary software for presenting these books already available, it is important to have a source code available implementation around to prevent inadvertent commercialization of the standard.

3. Write plugins for various browsers. Initially plugins that act like an embedded PDF reader and display only SMIL documents would the goal. Then integration with the main rendering engine for displaying XHTML+SMIL and SVG+SMIL documents (which is a much harder job).

4. Implement any missing pieces of the SMIL Animation Recommendation and the SMIL DOM interface. These two documents from the W3C provide extensions to the initial SMIL 2.0 specification. Extending `libsmil` to cover these features should not be too much of a stretch.

8 Playing With `libsmil`

The `libsmil` implementation discussed in this paper can be downloaded from the GNOME CVS repository (see [3] for instructions if you are unfamiliar with accessing that repository). The code is in the `smil` module, which contains the library as well as a few small applications and extensive documentation for library hackers and client developers alike.

Once the library has stabilized a little more, tarball releases will be made and the download site posted in a few popular locations.

References

[1] ANSI/NISO Z39.86-2002 Digital Talking Book standard `http://www.niso.org/standards/resources/Z39-86-2002.html`

[2] DAISY 2 Digital Talking Book standard `http://www.diasy.org`

[3] The GNOME CVS repository `http://developer.gnome.org/tools/cvs.html`

[4] SMIL 1.0 specification. `http://www.w3.org/TR/REC-smil/`

[5] SMIL 2.0 specification. `http://www.w3.org/TR/smil20`

[6] The Synchronized Multimedia group at the W3C. `http://www.w3.org/AudioVideo/`

[7] The Timed Text group at the W3C. `http://www.w3.org/AudioVideo/TT/`

[8] XHTML+SMIL—a W3C Note. `http://www.w3.org/AudioVideo/TT/`

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*