

The SMIL 2.0 Timing and Synchronization model

Using Time in Documents

Patrick Schmitz

January 2, 2001

Technical Report
MSR-TR-2001-01

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

<i>Amazon v. Audio Pod</i> US Patent 10,805,111 Amazon EX-1070

Abstract

Time is at the heart of multimedia, media centric documents and animated presentations. A powerful, flexible model is needed to unify scheduling, interaction, advanced control for animation, and runtime synchronization management. SMIL 2.0 defines a language and semantic model that addresses these needs. Its declarative language for timing and synchronization provides significant advantages over a purely programmed or scripting model. The new model has also been structured to facilitate integration of the XML syntax for timing and synchronization into other XML applications. The powerful model, declarative syntax and ease of integration make SMIL 2 timing and synchronization a good choice for document applications that include timing, scheduling and/or interaction. This paper introduces the syntax and the underlying semantic model. It discusses the advantages of a declarative approach, and the approaches for integration in other XML languages.

Keywords: timing, synchronization, time-based scheduling, multimedia documents, multimedia languages, synchronized multimedia, SMIL.

Introduction

The World Wide Web is becoming ever more dynamic, competing for audiences used to the effects and visual quality of television. This leads to a desire for more action and interaction in web pages, including richer media and animation. However, if the cost of authoring this richer content is too high, it simply will not happen. Traditional web browsers supported media with plug-ins, "helper" applications and a good deal of script. To make the authoring of multimedia simpler and more consistent, the W3C created the SMIL 1.0 language [SMIL1]. An updated version of this language is in final review as this is published - this updated version of SMIL will provide authors with the tools they need to author web pages with rich media, animation and interaction.

At the heart of this discussion is the representation of *time* in documents. We all have a visceral sense of time. Days pass into nights and days again, seasons come and go, and we grow older (whether we like it or not). Time is central to our existence - so central that we often do not separate it from simply "being". One of the main ways we distinguish sleep from our waking existence is by the lack of any time sense, as though time speeds up or jumps forward from the point at which we fall asleep to the time we awake. We understand and describe much of our experience in terms of time. For example, *movement* is change of position over time, and growth is a change of size or maturity over time.

In theory then, we should have no trouble describing and understanding a model for timing and synchronization. Unfortunately, the opposite is often true! In fact, our very familiarity with time leads us to believe that we all understand and share a common, solid model for time. Software developers and authors commonly assume that they need not study time to represent it in their designs, their programs and even in their documents. This assumption often leads to problems when *time* as they have modeled it does not behave the way they would like. Simple models for time often break down when the use-cases go beyond the trivial. These models often do not allow an author to describe more complex relationships among elements, or to easily model and control the behavior of elements over time.

While time is essential to media scheduling, a general time model must go beyond simple scheduling such as that provided in SMIL 1.0. Media schedulers rarely integrate user interaction in the time model, but modeling user interaction and other events as a form of scheduling provides a unified timing model for these commonly disparate features. Also, animation frameworks such as those used in graphics and presentation applications are based on time, but require additional features not commonly found in media schedulers. Finally, fixed scheduling models are too brittle to accommodate unreliable delivery, as is common on the Internet.

The SMIL 2.0 model for timing and synchronization addresses these issues, providing a powerful, flexible and extensible framework for representing time in documents. Its declarative language provides significant advantages over a purely programmed or scripting model. A modular structure supports integration with other XML languages, allowing authors and tools developers to share a common model across different language applications. The rest of this document presents an introduction to the SMIL Timing and Synchronization model and its applications. A language overview and discussion of integration issues are followed by a syntax summary and some examples. The bulk of the paper details the underlying model, and finally I present the motivations for a declarative language.

What is SMIL?

SMIL ("Synchronized Multimedia Integration Language") is a language for describing multimedia documents that can be viewed in a web browser. The first version, SMIL 1.0, was formally released by the World Wide Web Consortium in June of 1998, and a

second version, SMIL 2.0 [SMIL2], should be nearing completion before this is published. SMIL is based upon XML [XML], as are most of the current and next generation web languages (e.g. even HTML has been recently reworked to be an XML language, called XHTML). Like HTML, SMIL has *elements* (a.k.a. "tags") that declare content in the document, including video, audio, images and text. SMIL also has *attributes* that are used to describe how and *when* to make the media appear. Unlike HTML, SMIL does not concentrate on describing the actual media, but instead describes how all the media fit together into a presentation that plays back over time. Where a long HTML document would be navigated and read by *scrolling* from the top to the bottom, a SMIL document is more like a movie or presentation that is navigated in time and *played* from beginning to end.

SMIL is also a set of semantics describing how the language features work together. These semantics guide implementers as well as authors in the use of the language. The semantics provide a framework for extensions, so that the timing model and the animation model, for example, can be used as a basis for further development.

SMIL 2.0 is organized as a set of modules that group different areas of functionality. Each module has a set of elements and attributes (i.e., the *syntax*) used to describe the functionality as well as the associated *semantics*. The modules also define interdependencies, and any integration requirements for using the modules. The modules are:

Timing and Synchronization

Considered the core of SMIL, it is the part of the language used to describe *when* to make things appear and play and *how long* media should play. Also provides grouping to play media in a coordinated manner - for example in *sequence*.

Time Manipulations

Advanced timing features that provide additional control over the *pacing* of elements. Especially suited to animation use-cases.

Animation

Provides a declarative means of changing document values over time. It defines several elements that will set or interpolate properties and attribute values of other elements in a document. It also defines a model for *composing* the effects of multiple animation elements.

Media Elements

Provide a means to easily include video, audio, images, etc. in a document.

Transitions

Provide a simple means of declaring timed transitions (fades, wipes, etc.) on elements in a document. The transition model is based upon the timing and animation semantics.

Layout

Provides a means to describe *where on the page* to display media. The model used is somewhat like the absolute positioning feature of CSS, where each element is assigned to a particular rectangle on the page. However, it is quite different from the traditional layout model for HTML, in which content is laid out automatically in a *flow* across and down the page, according to the current web browser display area.

Linking

Provides much the same kind of functionality for hyperlinks that HTML supports, but adds a twist: when the user clicks a hyperlink into a SMIL document, the document timeline advances forward (or backward) to ensure that the referenced element is active (i.e. displayed or playing).

Content Control

Provides mechanisms to select or filter content within the page based upon various test criteria, such as the user's language and/or accessibility settings, the display characteristics, etc.

A few miscellaneous modules support language definition. For details see [\[SMIL-MOD\]](#).

The modularization of SMIL 2.0 is a significant advance, as it allows the SMIL language and semantics to be easily integrated into other XML languages. This flexibility was leveraged by the developers of SMIL 2.0 at the W3C, in the definition of two language variants. One language is based only upon modules defined by SMIL 2.0, and is referred to as the "SMIL Language" [\[SMIL2-LANG\]](#). The other language integrates SMIL 2.0 modules with XHTML modules, and is called the "XHTML+SMIL Language" [\[XHTML+SMIL\]](#). The two languages share the syntax and semantics for much of SMIL 2.0, providing authors and tools developers a common model to learn and support. There are two main differences between the two languages: the layout model and the support for HTML content as *media*.

The SMIL Language defines a layout model that is independent of HTML and CSS, and that is geared toward simple, rectangular design schemes. SMIL Layout has no support for text flow, tables and many other tools familiar to HTML authors. The XHTML+SMIL Language preserves the layout and

text flow tools of HTML and CSS, in place of the SMIL Layout module. In addition, the XHTML+SMIL Language models HTML elements as media in the same way that video, audio and images are used in the SMIL Language. This means that SMIL timing can control how and when a `<p>` (paragraph) or a `<div>` element is visible, or when a `<b>` (bold) element applies its associated effect.

The integration of SMIL functionality with XHTML and CSS is an important new tool for authoring. It allows the vast majority of web authors already familiar with the lingua franca of the web to take advantage of the multimedia support in SMIL, without giving up the tools they use every day. Where authors were forced in the past to place elements like video and animation in "black box" plug-ins, largely isolated from main page content, XHTML+SMIL allows authors to easily declare and control time based media, and to coordinate the HTML presentation in synchronization with their videos and animation. Authors will not have to choose between proprietary, media centric tools on the one hand, and the standardized, web centric tools of HTML, XML etc. on the other.

Integration overview - what does it mean to time HTML or XML?

I have described the SMIL modules, and emphasized the importance of this for integration of SMIL syntax and semantics with other languages. This begs the question: what does it mean to apply timing semantics to a markup language like HTML, and to the presentation semantics of CSS? For the SMIL media elements, the definition of the "begin" and the "end" of an element is simple and obvious: it controls when an element is displayed and (for continuous media like video and audio) when it begins to play. However, when timing is applied to elements in a language such as HTML, there are several possible interpretations for "begin" and "end", with different side effects in the document.

The easiest or most obvious integration arises for elements with an obvious (or clearly defined) "intrinsic" behavior. These elements may not describe media but commonly have some presentational effect. For example, the HTML elements `<b>` and `<i>` have the intrinsic meaning of "bold" and "italic" respectively. When timing is added to these elements, it clearly should control the presentation effect, causing the content to be "bold" or "italic" over time. The more abstract HTML elements like `<strong>` and `<em>` (for

"emphasis") do not have a specified presentation behavior, but they do have a clear intrinsic semantic that can be controlled (applied) over time. In general, an intrinsic behavior can be defined for XML language elements that represent *qualities* or *behaviors* (either in the abstract, or specifically presentational). Those elements with a clear "intrinsic" behavior have a natural semantic when timed.

However, many languages include elements that describe content that has no clear intrinsic behavior. Many of the content declaration elements of HTML (`<p>`, `<span>`, `<div>`, `<li>` etc.) fall into this group. Most XML language elements that declare *objects* or *entities* (as opposed to *qualities*) also fall into this group. The most obvious behavior is to show and hide the associated content over time, but even this can have several interpretations. HTML and CSS are based upon a flow-layout model in which the position and size of one element can affect the position of other elements in the document. Sometimes an author may wish to show and hide an element over time without affecting the rest of the document - this is described as controlling the element's *visibility*. Other times the author may wish to *reflow* the document when the element is shown and hidden, so that the effect is as though the element were actually removed from the document when it is hidden and "re-inserted" when it is active and displayed - in CSS, this is achieved by manipulating the *display* property for the element. In the general case, an XML element may either be removed entirely from the semantics of the document, or it may simply be hidden or disabled in some manner - the possibilities for a given language may be numerous.

The need to let the author describe the desired behavior led to the notion of a *time action*. The `timeAction` attribute in SMIL 2.0 lets the author specify one of a set of alternative "actions" to apply while the element is active in time.¹ In addition to the "intrinsic", "visibility" and "display" behaviors described above, other time actions were added including the ability to

¹ The SMIL 2.0 Language (as distinct from the XHTML+SMIL Language) does not include the module that defines the `timeAction` attribute (each SMIL language variant uses a different subset of the SMIL modules). The SMIL 2.0 Language only declares content using the SMIL media elements, and for these it effectively defaults the `timeAction` to "intrinsic". SMIL Layout is not based upon a flow model, and there is no support for CSS styles or class selectors in the SMIL Language, so the other `timeAction` values would not apply. The two languages are semantically compatible, but XHTML+SMIL just has a richer set of controls in this respect.

add a "class" definition to an element over time and the ability to apply an inline *style* definition to the element over time. The set of time actions legal in a given language depends upon the semantics of the language - the *timeAction* model can be extended to integrate SMIL 2.0 timing with any XML language.

The *host language* (the primary XML language with which SMIL 2.0 timing is being integrated) ultimately defines the semantics of the integration. The *timeAction* attribute provides the needed flexibility for a wide range of element semantics. Nevertheless, in some languages, not all elements may support timing. In addition, an author may selectively apply timing to some elements and not to others. In the general case, the time model refers to *timed elements* as those elements of the language that can support timing and have been specified as timed in the document. The time model co-exists with and interacts with the semantic model and Document Object Model for the XML language. Timing is separate from both the content and the presentation of the document, although it interacts with both.

What is "Timing and Synchronization"?

Timing and Synchronization is really about *choreography for documents*. Whenever a document has a notion of time as part of the presentation, and especially when a document incorporates time-based media like video and audio, there is a need to choreograph how all the pieces fit together, and to describe how they should be played. *Timing and synchronization* provide the tools to do this. SMIL 2.0 Timing and Synchronization also supports interaction and advanced timing for animation. Interaction is essential to the web medium, and the SMIL 2.0 timing

model ensures that media scheduling and interactive timing are unified, and not two separate worlds. The SMIL 2.0 Time Manipulations module integrates support for common animation use-cases in a centralized, integrated manner. The next sections will introduce the syntax for SMIL 2.0 Timing and Synchronization and describe the semantic model that underlies the language.

Syntax summary and basic examples

SMIL timing and synchronization describes the set of relationships and semantics that control *when* each element *begins* (e.g. when it appears or starts to play), the *duration* of the element (how long it plays), whether the element should *repeat*, and how it should behave when it *completes*. These features are all described with attributes that can be added to any element. The language makes it possible to make an element begin or end at a simple *point in time* (e.g., 10 seconds after the document begins), or when another element *begins* or *ends*, or when some *user-interaction event* is observed (like a button click), along with a few more obscure possibilities.

The basic syntax for timing and synchronization is quite straightforward. All timed elements have *attributes* that describe the begin time and the duration. This duration is referred to as the *simple duration*, because it defines a simple playing of the element - e.g. once through video, audio or animation. Additional attributes cause this simple duration to *repeat* and to obey additional constraints on the *active duration*, which is the total period that the element is playing, including any repeats. *Time containers* are elements that group other timed elements together for coordinated play. As timed elements, time containers also have begin and duration attributes, can repeat, etc. The complete syntax is in the specification [[SMIL2-](#)

Example 1

```
<html>
<head>
<style>  <!-- Bind the IE timing behavior to the needed elements -->
  t\:*, span { behavior:url(#default#time2); }
</style>
<XML:NAMESPACE PREFIX="t"/>
</head>
<body>
<t:audio begin="1s" src="media/talk.wma" syncBehavior="locked" />
<p>
  <span begin="1s">SMIL Timing syntax consists of a set of </span><br>
  <span begin="4s"><em>attributes</em> for controlling the behavior of media, </span><br>
  <span begin="7s">and several types of <em>time containers</em></span><br>
  <span begin="9s">that group media together for coordinated presentation.</span>
</p>
</body>
</html>
```

[TIMING](#)], but some examples illustrate the basics of the language.

The examples use the XHTML+SMIL Language. The associated demos require Microsoft Internet Explorer version 5.5 or newer (see [\[MSIE\]](#)). The discussion assumes basic familiarity with HTML [\[XHTML1\]](#) and CSS 2 [\[CSS2\]](#). While this document is not intended as a complete tutorial for SMIL authors, the examples should provide a reasonable introduction to the timing syntax. The demos are also available at: <http://research.microsoft.com/~pschmitz/demos/SMILTimingForTheWeb-Demos.html>.

Example 1 - declaring media and using begin

The video and audio elements (and a few synonyms) make it easy to declare timed media. A "src" attribute just points to the URL of the associated media.

The begin attribute lets the author control when an element is displayed, or begins to play. The attribute supports a wide range of different values, but the most commonly used values are simple offset times. The attribute value is just a number and a units indicator ("s" for seconds, "min" for minutes, etc.) as shown in

example 1.

The "syncBehavior=locked" attribute on the audio makes the presentation wait until the audio is really ready to start playing, before it can proceed. This ensures that even if media is slow to buffer (e.g. on a slow net connection), the presentation will remain in sync. This is further described below.

Example 2 - Adding durations, timeAction and Time containers

Now that we can control *when* something begins, we want to be able to control *how long* it lasts, and more specifically how timing *affects the element*. The dur attribute specifies the simple duration for an element - how long it plays or is applied. The value is just a number with units (as for begin offsets). The timeAction attribute controls what timing does to the element, as described above. In example 1, the default value "visibility" was applied; example2 specifies a "highlight" class for the timeAction.

SMIL 2.0 *time containers* make it possible to group a set of media elements in time, and control how they play back together. These include

- o a generic, *parallel* element **par** that will play the

Example 2

```
<html>
<head>
<style>
  <!-- Bind the IE timing behavior to the needed elements -->
  t\:*, p, span { behavior:url(#default#time2); }
  .highlight { background-color:yellow; font-weight:bold }
</style>
<XML:NAMESPACE PREFIX="t"/>
</head>
<body>
<!-- We wrap everything in a "par" so that it all starts together 1 second
      after the page is loaded. -->
<t:par begin="1s">
  <t:audio src="media/talk.wma" syncBehavior="locked" />
  <!-- We set a timeAction to apply a highlight to the spans.
        The p becomes a sequence time container for the highlights,
        and the spans just need durations -->
  <p timeContainer="seq" timeAction="none" >
    <span dur="2.9s" timeAction="class:highlight" >
      SMIL Timing syntax consists of a set of </span><br>
    <span dur="3s" timeAction="class:highlight" >
      <em>attributes</em> for controlling the behavior of media, </span><br>
    <span dur="2s" timeAction="class:highlight" >
      and several types of <em>time containers</em> </span><br>
    <span dur="4s" timeAction="class:highlight" >
      that group media together for coordinated presentation. </span>
  </p>
</t:par>
</body>
</html>
```

grouped media all together

- o a *sequence* element **seq** that will play the grouped media one at a time as a sequence
- o an *exclusive* element **excl** that will play the grouped media one at a time, but in any order

These same semantics can also be added to an element (like an HTML <div>), using the `timeContainer` attribute.

Just like grouping functions in a drawing tool, these time containers allow all the grouped media to be controlled as a unit. As a timed element, a time container also has begin and end times, a duration, repeat behavior, etc. This grouping model leads to a time containment hierarchy in the form a tree. The topmost time container is also the document root (the `body` element in SMIL and HTML). In example 2, a `par` time container delays the entire animation for a second at the beginning, and the HTML <p> element is given sequence semantics to make the highlights appear one after another.

The same principles shown in this demo generalize to a

whole class of applications, including slide-show presentations synchronized to audio and video, lyrics that have highlights displayed in sync with music, captions and subtitles displayed in sync with video or audio (including news, movies, etc.), and variants on the theme for education, accessibility, etc.

Example 3 - Adding repeats, interaction and sync-arcs

The first two examples show the basics of timing and synchronization. Next, we show how to make things repeat, how to add interaction, and how to specify additional synchronization relationships. Any element (including an entire time container) can be made to repeat the simple duration. This behavior can be specified as a number of times to repeat using `repeatCount`, or as a total duration to repeat using `repeatDur`. Example 3 repeats the main part of the presentation twice. Note that the duration of "main" need not be specified - it will be computed from the elements it contains.

To add interaction, an author can make an element begin or end in response to an event like a mouse click.

Example 3

```
<html>
<head>
<style>
  <!-- Bind the IE timing behavior to the needed elements -->
  t\:*, p, span { behavior:url(#default#time2); }
  .clickable { text-decoration:underline; background-color:cyan; color:navy }
  .highlight { background-color:yellow; font-weight:bold }
</style>
</head>
<body>
<!-- The paragraph shows up immediately, and stays until user clicks on the anchor.
      It restarts just after "main" ends, so the user can listen again. -->
<p begin="0; main.end" end="start.click" timeAction="display">
  <a id="start" class="clickable">Click here to begin.</a> </p>

<!-- The main part of the presentation will repeat twice, so readers
      really get the point. -->
<t:par id="main" begin="start.click" repeatCount="2" timeAction="display">
  <!-- We make the audio wait 1 second after "main" is displayed,
        and then we make the highlight sequence begin when the audio begins. -->
  <t:audio id="talk" begin="1s" src="media/talk.wma" syncBehavior="locked" />
  <p begin="talk.begin" timeContainer="seq" timeAction="none" >
    <span dur="2.9s" timeAction="class:highlight">
      SMIL Timing syntax consists of a set of </span><br>
    <span dur="3s" timeAction="class:highlight" >
      <em>attributes</em> for controlling the behavior of media, </span><br>
    <span dur="2s" timeAction="class:highlight" >
      and several types of <em>time containers</em> </span><br>
    <span dur="4s" timeAction="class:highlight" >
      that group media together for coordinated presentation. </span>
  </p>
</t:par>
</body>
</html>
```

The begin (or end) value specifies the *element* that will raise an event and the *name of the event* to respond to - for example `begin="foo.click"` will begin when the user clicks on the element with id "foo", and `end="img3.load"` will end when "img3" completes loading the associated media. The events may be user interaction events, system or media related events, or even custom streaming events.

In some cases, an author wants to describe a specific synchronization relationship between this element and another - this is called a *sync-arc*. The begin or end of one element is defined relative to the begin or end of another. An element can even combine multiple begin times, so that it can play more than once. Example 3 shows all these features.

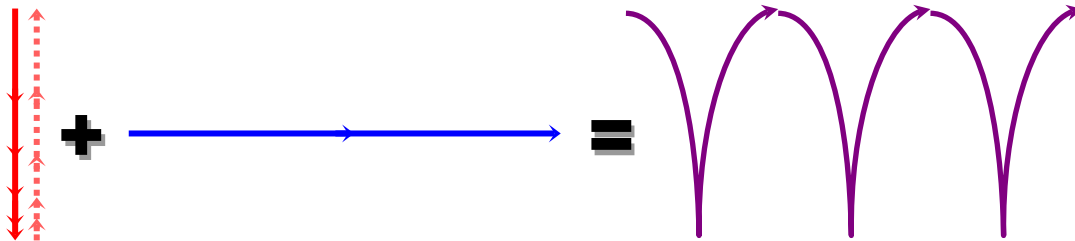
Example 4 - Animation and time manipulations

This example illustrates a common use of advanced timing features for animation. Several SMIL 2.0

elements make it easy to declare basic animation effects like motion, color interpolation, etc. (see also [\[SMIL-ANIMATION\]](#) and [\[SMIL2-ANIM\]](#)). The `animateMotion` and `animateColor` elements used in example 4 interpolate position and color values over time. To achieve a more realistic, mechanical feel to animation, the author must be able to describe acceleration and reversing (pendulum-like) behavior. A set of features called *time manipulations* provides this, allowing an author to control the *pacing* of an element. These are detailed later, but a simple description is provided to explain the example.

The `speed` attribute makes an element play faster or slower than normal - the value is a multiplier of normal play speed. The boolean `autoReverse` attribute causes the element to play forward, and then backward. The `accelerate` and `decelerate` attributes values control pacing during a *proportion* of the simple duration, so that the element speeds up from rest at the beginning, and then slows down to rest at the end. In the example, `accelerate` applied to the motion simulates gravity. The `autoReverse` attribute causes

Figure 1: Illustration of additive motion animation



Example 4

```
<html>
<head>
<style>
  <!-- Bind the IE timing behavior to the needed elements -->
  t\:*, h1 { behavior:url(#default#time2); }
</style>
</head>
<body>
  <h1 style="position:absolute; top:0; left:0" timeContainer="par" >
    Bounce
    <!-- Animate up and down (x=0, y=100), with accelerate to simulate gravity -->
    <t:animateMotion by="0,100" dur="3s" accelerate="1" autoReverse="true"
      repeatCount="indefinite" />
    <!-- Add a drift across the page (x=500, y=0) -->
    <t:animateMotion by="500,0" additive="sum" dur="18s" repeatCount="indefinite" />
    <!-- Fade color in and out in coordination with bounce -->
    <t:animateColor attributeName="color" from="white" to="gold"
      dur="3s" accelerate="1" autoReverse="true" repeatCount="indefinite" />
  </h1>
</body>
</html>
```

the animation to bounce up and down. The same attributes align the color animation to the motion.

The first motion animation creates vertical motion, accelerating down and then "bouncing" back up. The second motion animation adds in horizontal motion. It has a duration that is an even multiple of the vertical motion duration, so that they repeat in a coordinated manner. The effect of the motion is illustrated in Figure 1, and can be seen in the associated demo.

Summary

As the demos illustrate, the basic timing attributes control the begin, end and duration of elements, and repeat behavior. The `begin` and `end` attributes support a variety of ways to specify timing relationships between elements. The time containers `par`, `seq` and `excl` are used to group elements together and structure the timing relationships. The time manipulation attributes `speed`, `accelerate/decelerate` and `autoReverse` provide control over the rate at which time progresses, supporting common animation cases. The syntax is integrated directly into HTML, and controls both traditional "media" like video and audio, as well as the HTML content and CSS presentation. The `timeAction` attribute provides control over the behavior of timed elements, especially HTML content.

With these basics in mind, it is time to dive into the underlying model.

What's under the hood?

In addition to defining the *syntax* for all the features, the timing and synchronization model defines how the features are related and interact. It includes a *time graph* model for representing the relationships among timed elements, and for managing *dynamism* in these relationships. The time graph includes a tree structure that represents the time containment hierarchy, as well as links that describe synchronization relationships. The semantics are detailed in the following sections.

Background concepts

Several definitions, and some basics of timing must be understood at the outset.

The SMIL 2.0 timing modules define the means to coordinate and synchronize the presentation of *media* over time. In this sense the term *media* covers a broad range, including what SMIL calls *discrete* media that

have no intrinsic timing (such as still images, HTML, vector graphics, etc.) as well as *continuous* media types that are intrinsically time-based, such as video, audio, and animation. The distinction is important, as it affects the *implicit duration* of an element. Continuous media have an intrinsic duration associated with the media - SMIL uses this if the `dur` attribute is not specified. Discrete media have an implicit duration of 0.

In the model, time can also be thought of as "progress" from the beginning to the end of the media (or the animation, or the time container). The important thing is to remember that time in the model is not real-world time, and can differ in many ways - time in the model can be paused, resumed, repeated and sped up or slowed down.

The time model for an element is built up as a nested set of intervals within intervals. The *simple duration* is the innermost interval, and represents a unit progress through the media, animation, time container, etc. The effects of acceleration, deceleration and `autoReverse` are all applied to the simple duration. The simple duration can be repeated, and it can also be constrained with end values, and minimum and maximum durations. The effects of repeat and these constraints define the *active duration* for the element. The active duration for any element exists (only) within the simple duration for the parent time container, and is constrained to not extend before the beginning or after the end of the parent simple duration. When a time container repeats, all timed children will begin and end within each repeated instance of the parent simple duration. If time manipulations are specified for the time container, they will affect the pacing of the time container contents *as a unit* (this is detailed below). Time for any element is defined by the context of the element as well as the timing attributes defined on the element itself.

Arithmetic with time

Arithmetic functions for time are based upon a common model for time durations called *interval timing*. In this model, an *interval* corresponds to the duration of the element, but it has specific characteristics at the end points: an interval *includes* the begin time of the element, but an interval *excludes* the end time. That is, an interval of three seconds that starts at 0 includes the time 0, and all the times up to but not including the time 3. This is also referred to as *end-point exclusive* timing.

For programmers, this is essentially like 0-based indexing. Most media authors familiar with timeline editing tools will also recognize this model. For those unfamiliar with interval timing, a real-world example may help: Interval timing directly corresponds to the way that seconds add up to minutes, and minutes add up to hours. Although a minute is described as 60 seconds, a digital clock never shows more than 59 seconds. Adding one more second to "00:59" does not yield "00:60" but rather "01:00", or 1 minute and 0 seconds. The theoretical end time of 60 seconds that describes a minute interval is excluded from the actual interval.

Interval timing or end-point exclusive arithmetic is commonly used in media editing tools as well as media schedulers and animation engines. The important thing to remember is that something with a duration of 10 (in any units) is not rendered at time 10.² There are cases when an interval must be described as having no end-point - effectively lasting "as long as possible" (in the element context). The SMIL 2.0 timing model uses the special value "indefinite" to represent this. The rules for arithmetic with "indefinite" are much like the widely used rules for the common notion of "infinity". An "indefinite" value plus or minus any duration is still "indefinite", and "indefinite" compares greater than all other times.

Relative timing and time dependency

Although we talk about an element beginning and ending at particular times, within the model we always define an element's times relative to some other element, and not to some overall document timeline. When a simple offset - e.g. 5 seconds - is used to describe a begin or end time, the semantics of the parent time container specify which element the offset is relative to. For example, for a child of a `par` element, the offset is just relative to the begin time of the `par`, and for the child of a `seq` element, a simple begin offset is relative to the end of the previous child element. Other ways to specify begin and end values allow the author to describe times relative to specific other elements, to events, or even to a real-world *wall-clock* time. See the [specification](#) for syntax details.

² Interval timing requires careful evaluation in some instances. When the SMIL fill attribute is applied to an element to freeze the element at its end, the last state or value of the element is used. For animation the final value is used, although one might initially interpret the animation to define the ending value as happening at the end time, and not just before. Nevertheless, since the animation duration is also defined as end-point exclusive, the model is consistent.

All the elements in the time graph are connected through the time containment hierarchy. This means that if an element time *t* is defined relative to any other element, the time *t* can be converted mathematically to the equivalent time relative to any other element. A common implementation requirement is to convert times into *parent simple time* - that is, to the equivalent offset from the beginning of the parent time container's simple duration. This conversion makes it easier to manage the contents of the time container as a logical unit, and simplifies the interpretation of aspects of the timing semantics (such as lists of begin and end times, as described [below](#)). The [specification](#) provides algorithms for performing these time conversions.

It is possible to define a begin or end time for an element that is *before* the beginning of the parent time container - in this case the computed offset in parent simple time is *negative*. This does not mean that the element actually begins before the parent begins. Rather, it defines the synchronization relationship for the element relative to the parent. An element with a negative begin time will begin when the parent begins, but will start part of the way through the simple duration, as though it had begun at the earlier time. For example a video with begin offset of -2 seconds will begin playing when the parent begins, but will cut off the first 2 seconds of the video and play from 2 seconds on.

Since each time in the time graph is relative to some other element (or more specifically to the begin or end of some other element), we can follow the *chain* of relative time relationships that is used to compute each time. The chain starts with the particular element time, and leads to the root time container at the top - all times eventually chain back to the root time container.³ The root - and thus the timegraph - begins when the document is presented (e.g. when it is loaded into a browser). When an element **A** defines a begin or end time relative to another element **B**, element **A** is described as a *time dependent* of element **B**, and this relationship is modeled in the time graph. These time dependency links are in addition to the links that represent the time containment tree structure; the

³ There is an additional level of indirection in some cases. Wall-clock timing is nominally defined relative to a time that is external to the time graph, but in fact this is converted to a time for the root time container using the wall-clock time for the document begin. Event-based times are defined relative to an event that can happen at any point, but these too are converted from a system time to a time on the root time container. Once the time has been established relative to the root time container, it can be converted to a time relative to any element in the time graph.

combination of both relationships forms the complete time graph.

It is important to note that the timegraph is a dynamic model, and not a static view of a document. It is sometimes easier to think of a document as having fixed, computed times, and represented by a static time graph. However, this is too simplistic, and can be misleading even in simple cases. At any given point when a document is playing, a static graph can be modeled that describes the state of the presentation and its constituent elements at that point. However, as the document is presented, aspects of the graph can change in response to user interaction, unreliable media delivery over a network, or manipulation of the document through [DOM] interfaces. As an element in the graph changes, this can have side effects on other elements in the graph, and so any change can ripple throughout the graph, changing the way the document is presented. The timing and synchronization model describes how timing dependencies are maintained, how changes propagate through the time graph, and how semantics are re-evaluated as the graph changes.

In some cases there is not enough information in the time graph to compute the value of a time, and the time is described as *unresolved*. The most common example of this is when a time is defined relative to some event like a user mouse click or key-press. Until the event actually happens, the time value cannot be known. An unresolved time may itself have time dependents (i.e. it may have other element times defined relative to it), and all these dependent times will also be unresolved. Once a time can be resolved (e.g. when a user actually causes the event to happen in the browser), the dependent times can also be resolved. The changing of times from unresolved to resolved is another source of dynamism in the time graph, and so often has side-effects throughout the graph.

Time containers

As mentioned above, SMIL 2.0 includes three types of time containers: `par`, `seq` and `excl`. The `par` and `seq` time containers existed in SMIL 1, and are fairly straightforward. The parallel time container `par` is the most general, imposing minimal constraints on the contained elements - basically that elements cannot play before the `par` begins, or after it ends. The sequence element `seq` imposes the most constraints, playing one element at a time in the order the child elements are declared. The child elements of a sequence allow a limited syntax for `begin`, only supporting a simple offset from the end of the previous element. This constraint ensures that the semantics of a

sequence are preserved (with begin times relative to events or other elements, children of a sequence could easily end up out of order or overlapping). While it is possible to describe a sequence structure using a `par`, it can be awkward in some situations and so the `seq` element is provided as an authoring convenience.

The *exclusive* time container `excl` is interesting primarily because of the *interrupt semantics* it introduces. `excl` does not impose an ordering on the contained elements as `seq` does, but like `seq` it only allows one child to be active at any given point in time. If one element is playing and another tries to begin, the interrupt semantics control what happens. Four behaviors are defined:

- the element that was already active is *stopped*, and the new element begins to play
- the element that was already active is *paused*, and the new element begins to play. When the new element completes, the paused element will *resume* from the point at which it was paused. This is useful for modeling cases like commercial insertions in a video program.
- the new element is *deferred*, and will not begin to play until the element that was already active has completed.
- the new element is *not allowed to begin* (i.e. no interruptions are allowed).

Additional elements and attributes allow authors to group media within the `excl` into different priorities, with specific interrupt controls set between priority groups. This allows sophisticated models, like that of television programming. Video "channels" could interrupt one another, stopping any active element. "Program" video would be paused when a "commercial" insert begins, and would resume when the commercial(s) complete. A commercial that tries to interrupt another commercial could be deferred until the current commercial completes.

These interrupt semantics have wider application than just the SMIL languages, and may form the basis of synchronization models for rich annotations of media like television and radio.

Hierarchic Time - the tree model

I have described how all the elements in the time graph are related through the tree model of time containment. What is important about the tree model is the way time is *transformed* from parent to child. The tree model can be compared to scene graphs in graphics models: At

each level in the tree, a transform is applied to convert from the parent view to each child view. In the case of timing, the "view" is how time proceeds for the element. Each element has a notion of when its *local* time begins - this is just an offset from when the parent begins. Each time an element repeats or restarts, its simple time begins again from 0. When a time container repeats or restarts, all of its children that were active are stopped, and the children will begin again (at their respective begin times) as though for the first time.

The time manipulations control the pacing of time for an element - that is, they *transform* time for the element. For time containers, this affects not only the element itself, but all descendent elements as well. This is an important aspect of the time model, and makes it much easier to describe sophisticated animations. The parent-to-child time conversion functions are somewhat complicated by the time manipulations, but not significantly so. For example, a *speed* applied to an element just scales the progress of time relative to the parent time container, and so an offset in parent simple time is multiplied by the *speed* value to compute an element time. The time manipulations specification [[SMIL2-TIME-MANIP](#)] details the modified equations for time conversions.

Sync-arcs and time dependents - the graph model

When one element's begin or end time is specified in terms of another element's begin or end time, this relationship is described as a *sync-arc*. It is modeled in the time graph as a time dependency, but it does not significantly change the model of hierarchic timing as described above. The sync-arc defines how to compute the time, but it does not define an ongoing synchronization relationship. That is, once the begin or end time for an element is computed, the element behaves the same as if its begin time had been specified as a simple offset from the parent time container begin. The sync-arc is a convenient way of describing a begin or end relationship, but it does not mean that the two elements will necessarily play together in sync.

SMIL 1.0 allowed element begin and end times to be defined relative to the begin and end of other elements within the same time container. These are sometimes called *short sync arcs*. SMIL 2.0 allows the sync-arc reference to be to *any* element that is not a descendent. A reference to an element in another time container is called a *long sync arc*. Other than the conversion from

one element local time to another element local time, there is little additional complexity in supporting long sync arcs over short sync arcs.

When the begin or end of the referenced element is not yet resolved (e.g. if the element begins or ends relative to some user event), it is not possible to compute the dependent time. Once the *sync-base time* is resolved, dependent times defined as sync-arcs can be resolved as well, and the equivalent begin offsets relative to their respective parent time containers can be computed. The computation requires converting the sync-base time to a *parent simple time*, as described above.

Mixing scheduled and interactive (event-driven) timing

SMIL 1.0 allowed authors to describe a *schedule* for multimedia, but provided no support for user interaction (other than hyperlink navigation). One of the important advances in SMIL 2.0 is the integration of interactive, event-based timing. The model does not separate interactive content from scheduled content, but rather extends the scheduling model to support event specifications for times. This represents an important evolution over the models in many earlier multimedia presentation engines. See also [[UNIFYNOTE](#)].

For authors, the integration is as simple as allowing an event to be specified in place of a fixed time or a sync-arc description. However, this required some changes to the model to define the associated semantics. The key change required was the notion of *unresolved* timing, in which a time existed logically in the time graph, even though it could not yet be calculated. Unresolved timing has many uses, and has come to permeate the semantics of the dynamic time graph.

The concept of unresolved times was implicitly present in SMIL 1.0 as well, although the timing model did not really describe them as such (the entire timing model was under-specified in the SMIL 1.0 recommendation). The case in SMIL 1.0 arose for media elements that had an unknown duration. For example many MPEG video renderers do not know the duration of the MPEG video until the entire file has been read. Since it is often desirable to stream the file as it is played, the time graph is active even though the video duration is unknown. For the purposes of the time graph, this is no different from an animation that repeats until an event happens, or an image that begins on an event.

For an event time, the only issue is how to convert the real-world time at which the event happens (e.g. when the user actually clicks on something), to an offset relative to the parent time container begin. This is accomplished by taking the real world time of the event, and subtracting the real-world time at which the document began. This yields a local time for the document root time container. The conversions described above yield a parent simple time as required.

When an unresolved time becomes resolved, all the elements defined relative to that time must be notified, so that their respective times can be calculated. This is referred to as *propagating time changes to dependents*. The same mechanism is also used in other cases as well, when a resolved time is changed from one value to another. This can happen in a number of ways, including when the media for an element is paused or cannot keep up with the defined synchronization because of network delivery delays, etc.

An important distinction between event timing and sync-arc timing is that an element is only *sensitive* to events when its parent time container is active, while sync-arcs have no such constraint. Thus an element defined to begin on some user event (such as clicking a button) will not "see" the click if its parent has not yet begun. In addition, if an element is defined to end relative to some event, the element is not sensitive to the event (and so the end time will not be resolved) unless the element is currently active when the event occurs. Among other things, this allows authors to create "toggle button" behavior by specifying the same event description for both *begin* and *end*.

Handling multiple begin and end conditions

The *begin* and *end* attributes can specify a *list* of times. This can be used to specify multiple "ways" or "rules" to begin an element, e.g. if any one of several events is raised. A list of times can also define multiple begin times, allowing the element to play more than once. This is not the same as *repeating* an element - a repeating element is not considered to *begin* again each time it repeats.⁴ The *begin* and *end* attributes control the *active duration*, and so multiple begin and end times can define *multiple active durations* for the

⁴ In addition, since the element exists in the context of a parent time container, the lists of times are evaluated in the context of each simple duration of the parent time container. When the parent repeats, or if it also can begin more than once, all the child elements are reset and the begin and end lists are re-evaluated.

element. Each of these active durations is described as an *interval* in the timing model. The interval model is actually the basis for maintaining time dependency relationships in the time graph, but the details are generally of interest only to implementers.

The *restart* attribute allows the author to control the behavior of an element with a list of begin times. It can be set to allow the element to restart at any point (*restart*="always"), to only allow the element to play once (*restart*="never"), or to only restart if it is not already in the middle of playing (*restart*="whenNotActive").

One of most powerful aspects of the multiple begin and end times arises when timing is defined with a *cyclic dependency*, as in this example:

```
<p id="red" begin="0; grn.end-2s" dur="4s" .../>
<p id="blu" begin="red.end-2s" dur="4s" .../>
<p id="grn" begin="blu.end-2s" dur="4s" .../>
```

These three elements will cycle around, each one coming in just before the end of the previous one. The definition of the begin time for "red" depends upon begin time of "green" (since the end of "green" is just the begin plus the duration, the end of "green" is computed from - i.e. depends upon - the begin). But the begin of "green" is defined to depend (again, indirectly) upon the begin of "blue", which in turn depends upon "red". The three elements define a mutual dependency. In the time graph, the time dependency relationships form a loop or cycle in this case, and thus the term "cyclic dependency" is used.

If the "red" element did not specify an additional begin value that is independent of the cycle, none of the three could begin - there would be no way to *resolve* the begin times. However in this case, "red" is also defined to begin at 0, and so has a resolved end time. This in turn means that we can resolve the begin time and end times of "blue" and thus "green". But as soon as we can resolve the end time for "green", we have a second begin time for "red", two seconds before "green" ends. This new begin time for "red" will start the cycle all over again. Many implementations that traverse the time dependencies in the time graph will use some form of recursion. When there are cycles in the graph, implementations must prevent endless recursion and yet still correctly propagate changes to times within the cyclic dependency. The timing model details a mechanism to effectively interpret the cycle one step (i.e. one full cycle) at a time, ensuring a reasonable and *implementable* semantic.

Controlling speed and pacing, especially for animation

Animation is a common application of timing. The recent integration of SMIL 2.0 timing with the Scalable Vector Graphics language [SVG] is a good example of the interest in this area. In the general sense, animation includes the time-based manipulation of basic transforms, applied to elements in a presentation. Some of the effects typically supported include motion, scaling, rotation, color manipulation, as well as a host of presentation manipulations within a style framework like [CSS2].

Authors often employ animation to model simple mechanics. Many use-cases are difficult or nearly impossible to describe without a simple means to control pacing and apply simple effects that emulate common mechanical phenomena. While it is possible to build these mechanisms into the individual animation behaviors, this requires that every animation extension duplicate this support, making the framework more difficult to extend and customize. Finally, an ad hoc, per-element model precludes the use of such mechanisms on structured animations (e.g. controlling the pacing of a time container of synchronized animation behaviors).

A much simpler model for providing the necessary support centralizes the needed functionality in the timing framework. This allows all timed elements to support this functionality, and provides a consistent model for authors and tool designers. The most direct means to generalize pacing and related functionality is to transform the pacing of *time* for a given element. This is an extension of the general framework for element time (sometimes referred to as "local time"), and of the support to convert from time on one element to time on another element. Thus, to control the pacing of a motion animation, a temporal transform is applied that adjusts the pacing of time (i.e., the *rate of progress*) for the motion element. If time is scaled to advance faster than normal presentation time, the motion will appear to run faster. Similarly, if the pacing of time is dynamically adjusted, acceleration and deceleration effects are easily obtained.

Three general time manipulations are defined, as mentioned in example 4 above:

accelerate, decelerate

A simple model is presented to allow acceleration *from rest* (i.e. where progress is not advancing) at the *beginning* of the simple duration, and/or

deceleration *to rest* at the *end* of the simple duration. The model is defined so that the length of the simple duration is not affected. The functionality is also described as "*Ease-In, Ease-Out*", and is commonly used with motion to make objects start and stop more realistically, and with other animations to give a smooth or mechanical feeling to the animation. The attribute values are expressed as a *proportion* of the simple duration, and take values from 0 to 1.

autoReverse

This causes the simple duration to be played once forward, and then once backward. This is used to model common mechanical phenomena that advance and reverse. Some examples include:

- o pendulum motion - a partial rotation that advances and reverses
- o pulsing effects - usually a scale transform, but sometimes an intensity or color change that advances and reverses
- o simple bouncing as in the animation example above.

speed

Controls the pacing (or *speed*) of element active time. The speed effectively scales the rate of progress of the element active duration, relative to the rate of progress of the parent time container. Values less than 0 cause the element to play *backwards*.

The `accelerate`, `decelerate` and `autoReverse` attributes apply to the simple duration; if these attributes are combined with repeating behavior, the effects occur within each repeat iteration.

When the time manipulation attributes are used to adjust the speed and/or pacing within the simple duration, the semantics can be thought of as changing the *pace of time* in the given interval. A mathematically equivalent model is that these attributes simply change the pace at which the presentation *progresses* through the given interval.

As the speed of time is transformed with any of the time manipulations, we adjust how document time is converted to element simple time. To understand this, think of the contents of an element *progressing* at a given rate. An unmodified input time value is converted to an *accumulated progress* for the element contents. Element *progress* is expressed as *transformed time*. If element progress is advancing at a constant rate (e.g. with a speed value of 2), the filtered time calculation is just the input time multiplied by the speed value. If the rate is changing (e.g. due to acceleration), the transformed time is computed as an *integral* of the changing rate (although because the rate of acceleration is constant, the integral reduces to a simple expression for the *average* speed). This effect is illustrated in Figure 2. The functions used to compute transformed time are detailed in [\[SMIL2-TIME-MANIP\]](#).

Note that when time manipulations are applied to time containers, the effect "cascades" to all descendents. For example, if "speed=2.0" is applied to a `par` time container, the effect is as though the entire contents of the `par` were "recorded" and then "played back" at twice normal playspeed. The observed speed for a given element is a function of the element speed as well as the speed of its ascendant time containers.

This model lends itself well to an implementation based upon "sampling" a timegraph, with *non-linear* media (also called "random access" media). Sampled models are commonly used in graphics and animation, for which "current" state can be calculated and rendered for any time. Some *linear* media players may not perform well with the time manipulations (e.g. video decoder/renderers that only work at normal play

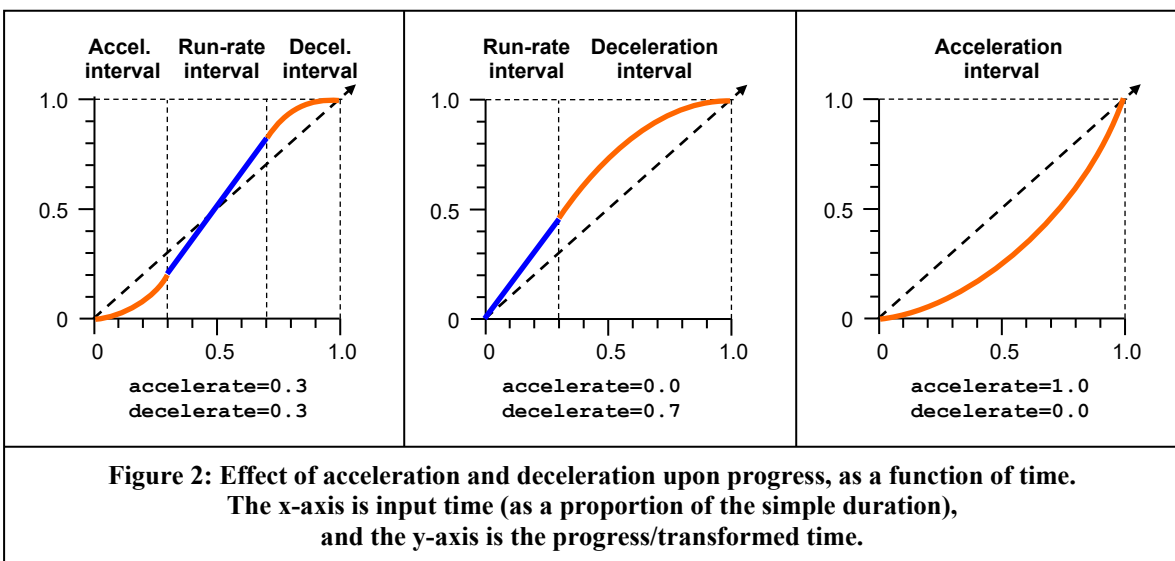
speed). A fallback mechanism is described in which the timegraph and syncbase-value times are calculated using the pure mathematics of the time manipulations model, but individual media elements simply play at the closest supported speed (which may be normal play speed).

Managing run-time synchronization of media

When multimedia documents are delivered over an unreliable medium like the Internet, the presentation often cannot perform as specified. In some cases media is not available for display when the associated element should begin. In other cases streaming media such as video or audio cannot keep pace or maintain synchronization as described. In the face of such problems, many simple models will just pause the entire presentation or will give up on synchronization when delivery problems occur. Neither of these approaches yields a very desirable presentation.

The SMIL 2.0 timing and synchronization model introduces a new model for controlling the run-time synchronization behavior of the presentation. Authors can control whether the document must wait for media elements that are not ready or cannot keep up, or alternatively, whether some media can be allowed to *slip* synchronization. Attributes on media elements are used to specify the media's synchronization behavior relative to its time container. When applied to a time container, the same attributes effectively control the *scope* of synchronization behavior. This is illustrated in example 5.

The video and audio will be held together in sync - if



Example 5

```
<body>
  <t:par syncBehavior="canSlip">
    <t:video src="somevideo.asf" syncBehavior="locked">
    <t:audio src="someaudio.wma" syncBehavior="locked">
  </t:par>
  ...
  <!-- The rest of the page content, including independent animation, etc. -->
  ...
```

either has to pause or "rebuffer" due to network congestion, the other will be made to wait. However because the `par` element is allowed to slip, the rest of the document will not be paused (or otherwise affected). The effect is that the author can define *scopes* of synchronization within the document. A time container that defines `syncBehavior="canSlip"` creates a new sync scope that includes all of its timed descendants. A time container with `syncBehavior="locked"` exists within the sync scope of its respective time container.

Another attribute lets the author designate a particular element as the *synchronization master* (sometimes described as the "clock source") for its sync scope. This is particularly useful for elements that represent broadcast media (such as television or live streaming sources), which cannot be paused or otherwise synchronized. It is also useful for media types that have hardware constraints, such as the inaccurate crystals on some audio boards. Many media players will "slave" video and other elements to audio. The `syncMaster` attribute allows the author to specify this type of behavior within a presentation.

Why a language and not an API?

You have seen the basic syntax for SMIL 2.0 Timing and Synchronization, and have been introduced to the concepts that underlie the timing model. By now you should have an appreciation for the flexibility and power of the language. But this begs the question: is a language for synchronized multimedia really necessary? Why not just code up some script that uses timers and the [\[DOM\]](#) to get the same effect? There are three primary reasons why a *declarative* language is better than a *procedural* approach. The first has to do with who authors the bulk of web content, the second has to do with how authoring tools work, and the third relates to adoption and fragmentation in the market.

Content authors are not programmers, and so script or code is not an option for them. If the approach to timing and synchronization requires a coded solution, it

generally requires that a programmer get involved in the authoring process. This makes the process much more expensive for content developers (paying two people to work on a given page, instead of just one, and coordinating their work as the page is edited and changed). In contrast, a declarative language is much more accessible to authors used to HTML and CSS, allowing the content authors to integrate synchronized multimedia without involving a programmer. The syntax makes it easy to hand-author many simple cases, with no knowledge of script, timers, callbacks and other programming paradigms. In a competitive marketplace, the lower cost to develop content is an important advantage for a declarative solution.

Authoring tool support requires a declarative solution. Although simple cases can be easily hand authored, authoring tools will be used to generate more complex or longer documents. In order to support timing semantics, these tools must be able to *round-trip* documents - i.e., to write out the document and then read it back in. This support makes it possible to create a document and then edit it later to refine it, to incorporate feedback, etc. The tools must also be able to exchange documents - i.e. read in a document that was edited in a different tool, or by some different author. If the timing semantics are represented in a *declarative syntax* like SMIL, the task to round-trip and exchange documents is vastly simplified - Specifications exist for the syntax in the form of Document Type Definitions and/or Schemas, and the semantics of the syntax are shared by all authoring tools and viewing platforms (e.g. browsers). In contrast however, if the semantics are represented in a procedural manner using a script or programming language, it is difficult at best to round-trip documents and often impossible to exchange documents. The problem is that there is no standard, agreed-upon way to program a given semantic. Each programmer or authoring tool will create some code for timers, for manipulating the document and for coordinating the media objects on the page. The script or code from one tool will generally vary from that in other tools, and so one tool will not recognize the timing script from

another tool, or script created by a programmer.⁵ To create an authoring tool that would recognize timing related script specifically is beyond current capabilities - it is a very hard problem to generally interpret the semantics of a block of code or script. Finally, script solutions will be fragile. If authors accidentally change the script, tools will likely fail to read it back in. A declarative language enables document round-trip and document exchange among authoring tools - a significant advantage over procedural solutions.

Declarative solutions will speed adoption and deployment in the marketplace. In order to achieve a given level of performance, the tendency of developers in the absence of a standard language will be to implement custom objects for video, animation etc. This means that authors will have to learn a variety of different approaches. Tools will fragment the authoring and viewing communities as they vie in the market. Because the proprietary solutions are designed as "black boxes" with respect to the rest of the web page, it is difficult (or impossible) to integrate different pieces of the puzzle. If one approach dominates, it may become a *de facto* standard, but intellectual property and licensing issues (among other things) will still limit the integration with other tools. We see this today with a popular commercial web-animation solution - the authoring tool is widely available, and the runtime "plug-in" (a very nice graphics rendering engine) enjoys broad deployment among web users. However, the animations (and the runtime) exist apart from other technologies like XML, HTML, CSS and associated document object models. Authors and publishers cannot easily incorporate animation into data-driven pages. They cannot easily adapt pages with this kind of animation for different types of platforms (as [XSL] and [CSS2] support). Without a standard, declarative solution, the authoring and publishing environment becomes fragmented, which ultimately hurts the adoption and advancement of technology. In contrast, a declarative solution based upon standard technologies ensures a cohesive and coherent authoring platform. At the same time, a standard declarative language does allow for fragmentation and competition among the

⁵ It could be argued that if a standard set of script methods and declarations were agreed upon, tools could exchange this script. I call this the "poor man's SMIL". Such a script-based "standard" would lack important advantages of a declarative language such as support for document validation (using DTDs or Schemas). In addition, script-based solutions will make documents larger (probably much larger), since the "runtime support" must be included with each document. While a standardized API is useful, it does not address the needs of authors or authoring tools. Any advantages an API may provide are generally available as an adjunct to a language, via DOM interfaces.

browser and tools vendors, which spurs adoption and leads to high quality presentation platforms.

In addition to the three primary reasons, other arguments can be made for declarative solutions. For example the presentation performance will be more robust and predictable, since a professionally developed user agent provides a much better basis than ad hoc code. The declarative approach is also more secure: Since the syntax is bounded, authors cannot create presentations that present a security threat on the end user's (viewer's) computer. In contrast, if authors must resort to a code model to use multimedia, there is more room to hide "Trojan horse" and related attacks within code that presents itself as multimedia presentation support. Finally, as the web evolves toward the "Semantic Web" described by Tim Berners-Lee and others, declarative solutions will enable intelligent services including information brokers, search agents, and information filters. If multimedia is to be fully included in this "Semantic Web", a declarative language is a prerequisite.

Clearly, the declarative solution of SMIL 2.0 has many significant advantages over procedural, code-based solutions. For authors and publishers, the declarative language is the best choice. In addition, because SMIL 2.0 is based upon standard tools like XML and is designed for integration, future work will broaden the applications of the language and model.

Looking ahead - Integrating SMIL 2.0 Timing and Synchronization with XML languages

The modular syntax of SMIL 2.0 allows for the integration of timing and synchronization markup with other languages. In SMIL 2.0, this integration is accomplished by mixing elements and attributes directly into the syntax for the host language. Both the SMIL 2.0 Language and the XHTML+SMIL integration language declare the timing as inline syntax added to media (or more generally *content*) declaration. However in the course of discussions on the requirements for integration, two additional approaches were identified, beyond just inline syntax:

1. **Styled timing**, in which CSS or XSL stylesheets are used to apply timing to a language.
2. **Timesheets**, in which timing is separated from both content and presentation (style) control.

SMIL 2.0 does not define these other approaches, mostly due to time constraints of the Working Group but also due to the range of open issues with styled timing and timesheets. Looking to the future, each of these three options for integration is needed for a number of applications, and with this comes the need to *combine* the approaches.

A unified model must be developed that allows authors and language designers to choose the most appropriate syntax, and to combine the different methods for a given document. This timing integration model must preserve common timing semantics, and should work well with other new document tools such as XSLT. While much work remains to be done in this area, a summary of the current thinking is presented here.

Inline Timing integration

Inline syntax provides a straightforward and easy-to-understand means of adding timing to a language. This approach is fairly well understood, and is documented in detail in the SMIL 2.0 specification. Even in documents that use styled timing and/or timesheets, the inline syntax can be useful to augment or override more general rules (e.g. in templates) defined with the other methods. This requires that there be a clear hierarchy among inline timing, styled timing and timesheets.

CSS provides a good precedent for such a hierarchy, defining ordering and priorities in the stylesheet cascade. While the specifics of the CSS model may not apply, the general approach is a good one. Given the many authors familiar with CSS, a unified timing model should align to the general model and principles of the cascade model. In particular, inline syntax should override styled timing and perhaps timesheets as well.

Styled Timing integration

Styled timing is well suited to documents (or document

sections) in which the document structure and the timing structure are fairly closely aligned. An example of this is list structures that are presented as a sequence. For example the HTML `<ol>` construct may be presented with a highlight style applied to each list item for a given duration, in sequence, as in example 6.

In order to combine styled timing with inline syntax, the semantics must be consistent between the two approaches. CSS again provides a good precedent for a unified model, supporting both stylesheets and inline style specification in the cascade. CSS 3 may provide support for setting properties in this way (possibly requiring namespace qualifiers), and XSLT can already. In any case, the unified model for timing integration should base styled timing upon the manipulation of the same set of properties that the inline syntax controls.

However, there is a problem with this model that must be addressed. A commonly used `timeAction` variant modifies the "class" property of an element (as in the example above). When this happens, a (new) style rule may be applied to the element that modifies the timing properties of the element. Another `timeAction` controls the application of an inline style rule, which can similarly change the timing properties on the element. In both cases, timing can cause the application of a style rule that in turn modifies the timing. This can easily produce non-obvious (even paradoxical) results, and may also lead to cycles in the evaluation of style rules. XSLT may not have this issue, as it does not support dynamic re-evaluation of selectors, but for CSS, this issue must be addressed.

A unified model should preclude this feedback complexity. One possible approach is to "lock" the timing properties while applying the time actions for an element so that the styling side-effects of the time action cannot feed back into the timing. Other approaches may also be explored.

Example 6

```
<style>
  ul { timeContainer:seq }
  li { dur:5s; timeAction:class\:highlight; }
  .highlight { text-decoration:underline; background-color:yellow; }
</style>
...
<ul>
  <li>My name is Norman Bates</li>
  <li>I'm just a normal guy!</li>
</ul>
```

Timesheets

Timesheets allow authors to specify timing and synchronization in a separate document, or in a separate section of a document. While this may sound like the separation of content and presentation that CSS provides, timesheets are actually quite different from styled timing. Timesheets may be based upon an animation model, in which multiple actions can be declared for an element, and may even overlap in time. The timesheet syntax might combine SMIL 2.0 timing elements and attributes with animation elements or "timeAction" elements that act upon elements in a given XML document. Another approach defines timesheets as filters, somewhat like XSL transform sheets. Where styled timing just manipulates timing properties, a timesheet defines timing relationships among actions. Timesheets thus separate timing and synchronization from both content and presentation.

Timesheets address authoring scenarios in which it is easier or simpler to separate the timing and synchronization description from the content and presentation controls. Timesheets must support:

1. Describing a timing structure orthogonal to the document structure. Not all sequences happen in lists, or fit into the XML document structure; inline and styled timing will be cumbersome for this.
2. Applying timing and synchronization to a document without modifying the original document. Copyright restrictions and the logistics of the document creation and editing process often require that the timing not be inline to the content.
3. Coordinating multiple documents. SMIL 2.0 can coordinate multiple media files, and XHTML+SMIL shows how timing can be integrated with other XML languages. These approaches must be merged and extended to allow references to multiple documents (including media files as well as fragments of documents like XHTML).
4. Defining multiple orthogonal time actions on an element. Inline and styled timing can only define a single `timeAction`.

The first three requirements are addressed simply by virtue of separating the timing and synchronization markup into the timesheet. The last requirement would be addressed by modeling timesheets as *animating* properties. That is, all time actions are simply manipulations of some property over time. The property can be an abstract like the intrinsic action of

an XHTML `<b>` (bold) element, or a specific property like CSS visibility. Multiple references to the same property on a given element are resolved as multiple animations of the property, according to the rules already detailed in [\[SMIL-ANIMATION\]](#). The animation model is analogous to the cascade rules for CSS, but is better suited to dynamic manipulation and composition. The animation model also builds upon the CSS object model when CSS properties are manipulated, and so provides a framework for mixing all three approaches to integration. While these examples use XHTML and CSS, timesheets will apply to the broad range of XML languages and to other style languages as well.

Conclusions

SMIL 2.0 defines powerful, flexible syntax and semantics for timing and synchronization of media. It combines support for traditional media scheduling and interactive timing to provide a state of the art timing model. The model scales well from basic presentations with simple syntax, to complex and finely tuned multimedia experiences that integrate a range of media, animation, and user interaction. Unlike other models, SMIL 2.0 is designed specifically for web delivery, and allows authors to manage the unreliable nature of the Internet, without sacrificing a structured scheduling model.

The declarative language SMIL 2.0 defines has significant advantages over procedural models, particularly for the vast majority of web authors who are not programmers. As a standard language, SMIL 2.0 facilitates both hand-authoring as well as tool-based authoring with easy document round-trip and interchange.

The XHTML+SMIL language combines SMIL 2.0 timing with the most widely used web authoring tools, HTML and CSS, providing authors a standard, easy to learn set of tools for creating the next generation of web documents. As web content moves more and more to XML, the built in support for integrating SMIL timing with new XML languages will ensure that authors have a consistent model to work to. Continuing work on the timing integration models will enable a broader range of application use-cases.

References

[CSS2]

"Cascading Style Sheets, level 2", Bert Bos, Håkon Wium Lie, Chris Lilley, Ian Jacobs. W3C Recommendation 12 May 1998. Available at <http://www.w3.org/TR/REC-CSS2>

[DOM]

"Document Object Model (DOM) Level 2 Core Specification", Arnaud Le Hors, et al. Version 1.0 W3C Recommendation 13 November, 2000. Available at: <http://www.w3.org/TR/DOM-Level-2-Core/>.

[MSIE]

Microsoft Internet Explorer (version 5.5 or later). Software available at: <http://www.microsoft.com/windows/ie/>. Documentation available at: <http://msdn.microsoft.com/workshop/author/behaviors/time.asp>.

[SMIL1]

"Synchronized Multimedia Integration Language (SMIL) 1.0 Specification W3C Recommendation 15-June-1998". W3C SYMM Working Group. Available at: <http://www.w3.org/TR/REC-smil>.

[SMIL2]

"Synchronized Multimedia Integration Language (SMIL 2.0) Specification W3C Working Draft 21 September 2000". W3C SYMM Working Group. Available at: <http://www.w3.org/TR/smil20/>.

[SMIL2-LANG]

"SMIL 2.0 Language Profile", Nabil Layaïda, Jacco Van Ossenbruggen. Available at: <http://www.w3.org/TR/smil20/smil20-profile.html>.

[SMIL2-TIMING]

"The SMIL 2.0 Timing and Synchronization Module", Patrick Schmitz, Jeff Ayars, Bridie Saccocio, Muriel Jourdan. Available at: <http://www.w3.org/TR/smil20/smil-timing.html>.

[SMIL2-TIME-MANIP]

"The SMIL 2.0 Time Manipulations Module", Patrick Schmitz. Available at: <http://www.w3.org/TR/smil20/smil-timemanip.html>.

[SMIL2-ANIM]

"The SMIL 2.0 Animation Modules", Patrick Schmitz, Aaron Cohen, Ken Day. Available at: <http://www.w3.org/TR/smil20/animation.html>.

[SMIL-ANIMATION]

"SMIL Animation", Patrick Schmitz, Aaron Cohen. Available at: <http://www.w3.org/TR/smil-animation/>.

[SMIL-MOD]

"Synchronized Multimedia Modules based upon SMIL 1.0 W3C Note 23 February 1999", Patrick Schmitz, Ted Wugofski, Warner ten Kate. Available at: <http://www.w3.org/TR/NOTE-SYMM-modules>.

[SVG]

"Scalable Vector Graphics (SVG) 1.0 Specification", W3C Candidate Recommendation 02 November 2000, Jon Ferraiolo. Available at: <http://www.w3.org/TR/SVG/>.

[Timesheets]

"Timesheets - Integrating Timing in XML", WWW9 position paper, Warner ten Kate, P. Deunhouwer, R. Clout. Available at: <http://www.cwi.nl/~lynda/www9/Timesheets.www9.html>.

[UNIFYNOTE]

"Unifying Scheduled Time models with Interactive Event-based Timing", Patrick Schmitz. Available at: <http://research.microsoft.com/~pschmitz/UnifyingNote.html>.

[WEAVING]

Weaving the Web, Tim Berners-Lee, Harper, San Francisco, 1999

[XHTML10]

"The Extensible HyperText Markup Language: A Reformulation of HTML 4.0 in XML 1.0". W3C Recommendation 26 January 2000. Available at <http://www.w3.org/TR/xhtml1/>

[XHTML+SMIL]

XHTML+SMIL Language Profile (Working Draft), Patrick Schmitz. Available at: <http://www.w3.org/TR/2000/WD-smil-boston-20000622/html-smil-profile.html>.

[XML]

Extensible Markup Language (XML) 1.0 (Second Edition) 6 October 2000, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler. Available at: <http://www.w3.org/TR/REC-xml>.

[XMLTime]

A Unified Model for Representing Timing in XML Documents", WWW9 position paper, Patrick Schmitz. Available at: <http://www.cwi.nl/~lynda/www9/TimingIntegrationPositionPaper.html>.

[XSL]

Extensible Stylesheet Language (XSL) Version 1.0. W3C Candidate Recommendation 21 November 2000. Sharon Adler, et al. Available at: <http://www.w3.org/TR/xsl/>