Towards SMIL as a Foundation for Multimodal, Multimedia Applications

Jennifer L. Beckham[*], *Giuseppe Di Fabbrizio*[**], *Nils Klarlund*[**]

[*]University of Wisconsin
Madison, WI 53706

[**]AT&T Labs – Research
180 Park Avenue, Florham Park, NJ 07932

jbeckham@cs.wisc.edu, {pino,klarlund}@research.att.com

Abstract

Rich and interactive multimedia applications, where audio, video, graphics and text are precisely synchronized under timing constraints are becoming ubiquitous. Multimodal applications further extend the concept of user interaction combining different modalities, like speech recognition, speech synthesis and gestures. However, authoring dialog-capable multimodal, multimedia services is a very difficult task. Fortunately, the W3C has sponsored the development of SMIL, an elegant notation for multimedia applications, which has been embraced by both Microsoft and RealNetworks. In this paper, we argue that SMIL is an ideal substrate for extending multimedia applications with multimodal facilities. SMIL as it stands is not a general notation for controlling media and input mode resources. We show that all what is needed are few natural extensions to SMIL along with the addition of a simple reactive programming language that we call ReX. Our language is designed to be maximally compatible with existing W3C recommendations through a generic event system based on DOM and an expression language based on XPATH. It is also designed to be simple so that the fundamental notion of seeking time (e.g. going backwards and forwards in presentations) is preserved.

1. Introduction

A multimodal application combines several modes of input such as speech recognition, keyboard, and pointing devices with several modes of output, visual display, text-to-speech (TTS), etc. Interactions with multimodal user interfaces provide rich and versatile communication with a computer. Such interfaces are extraordinarily hard to program or even specify. Thus, a common accepted terminology for such applications would not only further their understanding, but also promote their actual use in Web browsers and portable devices. Fortunately, under the auspices of the W3C, major players in this area, such as Microsoft, RealNetworks, and Macromedia, have agreed on a standard called **S**ynchronous **M**ultimedia **I**ntegration **L**anguage (SMIL, pronounced *"smile")* [3].

Although SMIL will allow media such as HTML extended with time concepts, PowerPoint slide shows, and Flash Player presentations to be described in one framework, SMIL does not integrate other modalities than mouse clicks for interactive presentations. In this paper, we outline a framework for doing so by (1) extending SMIL with some additional concepts and (2) suggesting the incorporation of ReX (**Re**active **X**ML), a reactive programming language based on XML standards. Our work is particularly suited for integrating speech recognition with multimedia applications.

Let us briefly discuss one such application. AT&T has developed a multimodal spoken dialogue system for accessing information in the AT&T personnel database, mVPQ (**m**ultimodal **V**oice **P**ost **Q**uery) [1][2]. The system allows a user to request information associated with an employee, ask for help, and request other actions such as calling, paging or faxing. Along with spoken dialog, mVPQ also provides a visual display, pointing device, and soft-keyboard. Together, these modalities allow for clear and simple interaction with the database. Ambiguities are easily clarified with a combination of visual and audio input.

Current interactive systems like mVPQ include a dialog manager and a natural language understanding modules to achieve effective interaction with a user. Dialog management for spoken dialog systems could use different techniques ranging from simple state machines to more advanced machine learning techniques. It is not the purpose of our proposed SMIL/ReX integration to replace the dialog manager. Rather, our framework allows a dialog manager to describe concisely the media relationship within each dialog step in precise and flexible ways.

For those purposes, existing IVR-based standards, like VoiceXML [5], developed by a consortium of AT&T, IBM, Motorola, and Lucent, is not a suited notation with its simple-threaded execution model. VoiceXML is in some ways based on simplistic platform assumptions not present in SMIL; for example, in VoiceXML, the rendering of audio is done by queuing. This mechanism creates a conceptual problem: the program counter is not synchronized with the audio rendering. But it also creates a practical problem: there is little control possible of the rendering itself except for aborting it. Moreover, VoiceXML assumes no graphical support; in particular there is no integration of multimedia. We shall see how SMIL solves these problems, while allowing support for multimodal input through ReX.

2. SMIL and its Advantages to Audio Rendering

A SMIL document consists of *media elements* that render static visual surfaces or play audio or video. The temporal relationships among media elements are determined by *time elements* of which there are three kinds:

- **seq** elements execute their children in sequence;
- **par** elements execute their children in parallel; and
- **excl** elements execute one child at a time as in a state machine.

The beginning and end of time and media elements may be linked to the execution of other elements or to user interface events.

2.1. A Simple Example

To display the text *"listening!"* in an HTML **div** element named **prompt_area** so that the text is displayed every three seconds for a duration of one second, we may write

```
<SMIL:text id="listening"
     region="prompt_area" begin="0;
     listening.end + 2" dur="1">

   listening!

</SMIL:text>
```

Here **<SMIL:text> … </SMIL:text>** is an XML *element* named **text** (**SMIL:** just identifies it as part of SMIL). This element has only one child, which is a string *"listening!"* In general, an element may contain a sequence of child elements

and strings interspersed. The element also has *attributes*, which define name/value pairs. For example, the attribute **region** has value **prompt_area** (e.g. the name for the **div** container where the text is to be rendered). For the rest, **begin** specifies that the first rendering is to start at time 0 (relative to the start of the execution of the parent element) or two seconds after the element named **listening**, namely itself, stops playing; and the **dur** element specifies the duration of display to one second.

2.2. The Time Graph

The semantics of SMIL is explained in terms of a dynamically changing time graph that records relationships between events through what are called *synchronization arcs*. For example, a time element *X* can be set with an attribute **begin= "YName.end + 2"** to specify that it starts two seconds after another time element *Y* named **YName** has ended.

SMIL is designed to make such dependencies easy to express in a declarative way; similar effects in languages like Java or JavaScript would require tedious code for dynamically setting up pointer structures among threads.

The event delegation mechanism relies on explicit registration; reactive programming languages like SDL [7] and Esterel [6] have been designed to avoid such detailed programming. Instead, threads may directly observe events or signals generated elsewhere. It is sensible and interesting that SMIL has adopted a similar approach, through *synchronization arcs*. However, a traditional concept of event generation bubbling and handling is still needed for handling of user input.

2.3. Adding Speech Recognition

We now return to our example. While the *"listening!"* message is flashing, we would like to enable speech recognition with a grammar **names.grm** and render a spoken message. Obviously, these three activities are to take place at the same time, so we use the **par** element with three children, each child representing a thread of execution:

```
<SMIL:par end="asr.recognition">
  <SMIL:text id="listening"
       region="prompt_area" begin="0;
       listening.end + 2" dur="1">
     listening!
  </SMIL:text>
  <TTS:render>Who is the person you are
  looking for?</TTS:render>
  <ASR:listen src="names.grm"/>
</SMIL:par>
```

The activities so specified last until a **recognition** event is raised by the **ASR** module. In general, the **par** element lasts until all of the child elements have finished executing. In our case, the first child never finishes executing, since it keeps being rescheduled. Therefore, the **par** element together with all its children is explicitly forced to end on a recognition event. The **<ASR:listen …">** element is an abbreviation for handling an external resource. Note that the parallel element clearly indicates that barge-in is allowed. To disable barge-in, a **seq** element could have been used to play the prompt before the speech recognition is engaged.

2.4. SMIL for Temporal Control of Input Modalities

With more *control elements* that impose conditions on media or input modes while they execute, SMIL will allow us to very elegantly express temporal constraints on input modalities. For example, the element **<ASR:off/>** temporarily turns off speech recognition. We can then express that a rendering of a commercial **"cheap-longdistance.strm"** brought on an interactive radio service is (a) interruptible, but only after the first five seconds, and (b) lasts at most 15 seconds:

```
<SMIL:par end="asr.recognition" max="15s">
  <SMIL:audio src="cheap-longdistance.strm"/>
  <ASR:off dur="5s"/>
</SMIL:par>
```

Traditional IVR languages do not allow such control of resources, in part because of the audio queuing model.

2.5. SMIL for Pausing Presentations

The audio queuing model is also the reason why traditional systems cannot easily interrupt a presentation while making it possible to resume it later. Such a facility is desired for interactive radio, where a listener in an automobile may decide to temporarily pause the playing of a newscast in order to get local traffic conditions. SMIL allows for the concise expression of such scenarios through the **excl** element:

```
<SMIL:excl>
   <SMIL:audio begin="0"
   src="8oclocknews.strm" peers="pause"/>
   <SMIL:audio begin="weatherButton.click"
   src="localweather.strm" peers="stop"/>
</SMIL:excl>
```

Here, the 8 o'clock news is interrupted when the weather button is clicked. The **peers** attribute is **pause** to indicate that the newscast is not stopped when interrupted, only paused. So, after the local weather conditions are played the newscast is resumed from the place it was interrupted. (SMIL currently does not have facilities for expressing that the audio stream should be slightly rewound before being resumed).

2.6. SMIL for Prioritizing Aural Information

The **excl** element is more than a state machine: although only one child may play at a time, several other children may be queued. For example, the newscast is queued above when the weather button is clicked. In advanced audio-centric user interfaces, more than one audio stream may be queued at the same time. To reduce confusion, audio streams must be characterized according to priority and how they interrupt each other. SMIL offers an attractive conceptual framework for doing so: children of an **excl** element may be partitioned into *priority classes*. Each priority class is characterized by a **peers** attribute to determine how members of the class interrupt each other. The possible values are **defer** (don't interrupt, but put the interrupting peer into the queue), **pause**, **never** (don't start the interrupting peer), and **stop** (stop the current peer right away). A priority class may also determine specific policies for higher and lower priority elements.

2.7. SMIL for Skipping Backwards and Forwards in Presentations

SMIL already accommodates a feature often considered essential for interactive multimedia applications: the ability to rewind a presentation according to synchronization points. The emphasis of SMIL is on defining hyperlinks so that user of a media player may jump into the middle of presentations. But, as we shall see, the underlying concepts of history recording and *seeking* the timeline easily lend themselves to solving user interface issues for speech applications such as undo mechanisms.

3. ReX + SMIL for Multimodal Applications

ReX is an XML-oriented language. XML is a generic notation for labeled trees. Many XML technologies rely on XPATH [8], a simple expression language for testing and operating on numbers, strings, and trees. ReX is designed to promote the reuse of standard markup languages. It addresses the concerns of a multimodal dialog markup language with a platform that provides: (1) a simple XML target language with a minimal number of primitives; (2) an expressive thread control and event system that is able to handle future and current interactive languages like SMIL and VoiceXML; (3) a scalable and modular approach to external components like ASR, browsers, TTS, etc.; (4) a straightforward implementation and maintenance of the target language interpreter; (5) an

adaptation of XML languages XSLT [9] and XPATH where possible (XSLT is a general standardized language for transforming XML documents). Note that a ReX or SMIL program is itself an XML tree; also, ReX uses XML as a data type concept. ReX is centered around the notion of events. A ReX document defines a hierarchy of coroutines that define reactions to events and the dispatch of events. A document executes within one thread of control with a set of global variables that consist of name-value pairs. All communication among coroutines is through message passing. An event (message) consists of an XML tree value, a priority, a set of target coroutines described by an XPATH expression, and behavior attributes.

3.1. Why not ECMAScript (JavaScript)?

We have already indicated that a principal obstacle to using SMIL for multimodal applications is the lack of event handling and event generation capabilities. Of course, the traditional remedy is to resort to JavaScript or the standardized version called ECMAScript. Developed for DHTML (Dynamic HTML), JavaScript is also an essential part of VoiceXML, where a subset is glued onto the XML-based scripting language. Although ECMAScript is not a multithreaded language, it would appear possible to combine it with SMIL. However, for all practical purposes, it would be very difficult to imagine how reversible presentations could be achieved in such framework. The problem is that ECMAScript is based on a comprehensive, object-oriented storage model. Thus, taking snapshots of the store in order to implement the history mechanism of SMIL seems like a hard problem.

Also, ECMAScript is a cumbersome notation for even simple XML processing: while-loops are required for testing tree properties, no standard serialization is provided, and programming is pointer-oriented.

3.2. Event Generation and Catching

The programming of IVR systems, even at the highest user interface level, is heavily centered on event handling. For example, VoiceXML provides a mechanism of defining **`help`** events that can be raised at any time during execution. Such events are caught in a closest enclosing block defining a handler; "closest" means closest to the point where the program pointer is when the event is raised. Rex defines a similar mechanism through a primitive **`<raise target="...">`**`...` **`</raise>`** where the **`target`** attribute specifies the destination of the event and where the content of the raise element is an arbitrary chunk of XML that is the value of the event. The name of the event is identified with the name of the root element of the value.

Events are caught in an *await* statement, which is somewhat similar to a try statement of Java or ECMAScript exception handling. Each child of an **`await`** statement is an event handler. The event handler is able to catch events raised anywhere within the await statement. The first child may be an ordinary statement; it is executed in the absence of events. As with the SMIL **`excl`** element, at most one child is executing at a time. The meaning of raising an event in a thread is the conventional one: first, the event handler is located that is syntactically closest to the current program counter location of the thread; second, the program counter is transferred to this event handler.

Below, we illustrate how a speech recognizer may be controlled using events. The first part of the await statement sends a grammar specification to the recognizer; then the indefinite duration suspends the execution. When the speech recognizer sends a **`nomatch`** event back to the ReX program, the **`handle`** element is executed.

```
<ReX:await>
 <ReX:raise target="id(asr)" dur="indefinite">
  <add-grammar src="names.grm" generate="help"/>
 </ReX:raise>
 <ReX:handle event="ASR:nomatch">
   <TTS:render>We didn't understand you.
   </TTS:render>
 </ReX:handle>
</ReX:await>
```

The **`<TTS:render>`** `We didn't understand you.` **`</TTS:render>`** statement is an abbreviation for a little piece of code that controls the text-to-speech component:

```
<ReX:await>
  <ReX:raise target="tts" dur="indefinite">
    <queue>
      We didn't understand you.
    </queue>
  </ReX:raise>
  <ReX:handle event="TTS:done">
    <ReX:exit/>
  </ReX:handle>
</ReX:await>
```

3.3. External Components

To control input modes, ReX communicates with external components through the event mechanism. For example, to use speech recognition input, we declare

```
<ReX:use id="asr" href="..."/>
```

The effect of this declaration is to set up a communication link between the resource declared in the **`href`** attribute. We envisage that standardized components be declared. For example, the ASR module may accept an event **`add-grammar`**. It may also declare that it may post events like **`recognition`** (for arbitrary parts of the Rex program to listen to), or it may declare events that are raised in the client application such as **`nomatch`**.

3.4. Focus Definition

In a dialog system, there is a natural notion of *focus*: it is what determines what can be said and the meaning of events such as **`help`**. SMIL does not provide such a notion; in fact, it does not even provide a notion of keyboard focus. We suggest that program variables are used to control the focus; our main example declares one such variable, called **`focus`**. The name of the variable is passed to the ASR component when it is initialized so that speech recognition events go to where **`focus`** direct it.

3.5. Reversible Computational Notation

A SMIL/Rex program is not meant to be computationally expensive. Typically, it will have a handful of variables at most. XML values that are received from outside components are sent along with little or no modification. Although looping behavior of programs between events is possible, such programming is meant to be restricted to simple string processing and searches in trees. Thus, the state of the program does usually not change much between events. And, in this way, ReX can be added to SMIL while keeping the history mechanism essential to time seeking.

3.6. Checkpoint-based Resumption

To allow seeking backwards in time, a SMIL implementation must rely on a history mechanism that record past several events. We suggest that SMIL be augmented with a *resume* mechanism that allows programmer specified temporal marks to be specified. These resumption marks indicate an interesting point to skip back to. A rewind button may then seek the presentation to the last occurring mark. Similarly, if a presentation is interrupted, then a later resumption should start from the last occurring mark. Concretely, we envisage that a section of a program using this mechanism is marked by an attribute

3.7. Event-triggered Resumption

Our last contribution to the SMIL model addresses a deficiency of traditional programming constructs to deal neatly with temporary interruptions such as a request for help. We have already discussed the possibility of modeling such interruptions through an event concept. For example, in VoiceXML, a help handler can be specified at the document level. It may catch help events for deeply enclosed parts of the dialog contained within some block without - and this is somewhat surprising - transferring control out of the block. This explanation is arrived at by assuming that the help handler is syntactically copied onto all inner blocks not themselves providing an explicit handler. The gist of the approach is that event handling does not transfer control out of the interrupted task so that any event happening during the interruption is processed as if the interruption had not occurred. A more substantial problem with this approach is that there is no notion of resuming the interrupted task. Instead, it must be restarted under programmatic control. For example, in VoiceXML there is no way to prevent a long prompt to be replayed from the beginning after an interruption.

Both problems are addressed by the SMIL **excl** concept. However, we need to extend the SMIL model in one direction: SMIL assumes that an interrupted task can be resumed only if the task that interrupted finishes execution. We need to add the possibility that events sent to an interrupted task will make it resume. With this solution, we have solved a semantic peculiarity of VoiceXML. Better still, this mechanism can be combined with event-based assumption to achieve media control not possible with VoiceXML such a resuming an interrupted newscast at the beginning of the last story.

4. The mVPQ Kiosk Example

We have put an example together based on experiences with the mVPQ kiosk application of AT&T. It illustrates speech input as an alternative modality that may be turned on by the user; synchronization of visual display, audio rendering, and speech recognition; and of most of the features of SMIL and ReX discussed above. A full example is beyond the scope of this paper. The snippet below shows the core of the interaction between SMIL and ReX.

```
… <!-- main thread -->
<SMIL:excl>   <!-- normal thread -->
   <SMIL:seq priorityClass="normal"
        resume="explicit">
<!-- throw input events through 'focus' -->
     <ReX:assign var="focus" expr="thread()"/>
     <!-- wait for user to activate ASR -->
     <SMIL:seq id="waiting"
          end="name_input.onchange">
        <SMIL:text region="prompt_area"
             end="speak_button.onclick">
          Push button 'speak' to speak, or use
          keyboard.
        </SMIL:text>
        <!-- turn ASR on -->
        <ReX:raise target="id(asr)"><turn-on
        through="focus"/></ReX:raise>
        <!-- enable help grammar -->
        <ReX:raise target="id(asr)"><add-grammar
        src="help.grm"
        generate="help"/></ReX:raise>
        <SMIL:par id="speech_on" resume="here">
          <TTS:render>Who is the person you are
          looking for?</TTS:render>
          <SMIL:text id="listening"
               region="prompt_area" begin="0;
               listening.end + 2" dur="1">
            listening...
          </SMIL:text>
          <ReX:await>
            <ReX:raise target="id(asr)"
            dur="indefinite"><add-grammar
            src="names.grm" generate="help"/>
            </ReX:raise>
            <ReX:handle event="ASR:recognized">
              <ReX:assign name="name_input"
              value="event()/result"/>
              <ReX:exit loc="speech_on"/>
            </ReX:handle>
            <ReX:handle event="ASR:nomatch">
              <SMIL:par>
                <TTS:render>We didn't
                understand you.</TTS:render>
                <SMIL:text
                region="prompt_area">Please
                say again.</SMIL:text>
              </SMIL:par>
            </ReX:handle>
          </ReX:await>
        </SMIL:par>
     </SMIL:seq>
<!-- $name_has received a value, submit it -->
     <ReX:submit
     next="process_name.cgi"><name><ReX:value-
     of
     expr="$name_input"/></name></ReX:submit>
   </SMIL:seq>
   <!-- help handler -->
   <SMIL:par priorityClass="instructions"
        beginHandler="help" dur="10s">
     <TTS:render>
       Please say the name of the person you
       are looking for. Or, type the name on
       the keyboard.
     </TTS:render>
     <SMIL:text region="help_area">
       Push button 'speak' to speak. Say the
       name of the person.
     </SMIL:text>
   </SMIL:par>
</SMIL:excl> …
```

5. Conclusion

We have argued that SMIL is an excellent basis for multimedia presentations augmented with speech recognition input. Specifically, we have suggested augmenting SMIL with ReX, a very small set of primitives from reactive programming languages; in order to get a very expressive and concise foundation for programming multimodal multimedia user interfaces. Along the way, we have indicated that VoiceXML is barely sufficient as the foundation for speech-centric multimedia applications. We believe our ideas hold the promise of creating a strong standardized foundation for multimodal, multimedia applications.

6. References

[1] Di Fabbrizio, G., et al. - *"Unifying Conversational Multimedia Interfaces For Accessing Network Services Across Communication Devices"*, IEEE International Conference on Multimedia and Expo, New York City, New York, USA, July 30 - August 2, 2000.

[2] Di Fabbrizio, G., et al. - *"Extending a Standard-based IP and Computer Telephony Platform to Support Multimodal Services"*, ESCA workshop on Interactive Dialogue in Multimodal Systems, Kloster Irsee, Germany, June 22-25, 1999.

[3] *"Synchronized Multimedia Integration Language (SMIL 2.0) Specification"*- W3C Working Draft 01 March 2001, http://www.w3.org/TR/smil20/

[4] *"Document Object Model (DOM) Level 2 Core Specification"*, http://www.w3.org/TR/DOM-Level-2-Core/

[5] *"Voice eXtensible Markup Language (VoiceXML™) version 1.0"* - http://www.w3.org/TR/voicexml/

[6] G. Berry and A. Benveniste - *"The synchronous approach to reactive and real-time systems"*, Proc. of the IEEE, vol. 79, n° 9, September 1991

[7] *"Specification and description language (SDL)"* - ITU Recommendation Z.100 (11/99)

[8] *"XML Path Language (XPath) Version 1.0"* - http://www.w3.org/TR/xpath.html

[9] "XSL Transformations (XSLT) Version 1.0" - http://www.w3.org/TR/xslt.html