

University of Technology, Sydney  
Faculty of Engineering

# IMPLEMENTING SCALABLE VECTOR GRAPHICS IN MOZILLA

by

**Brian Birtles**

Student Number: 00035927

Project Number: S04-096

Major: Software Engineering

Supervisors: Mr Steve Murray  
Dr Craig Scott

A 12 Credit Point Project submitted in partial fulfilment of the  
requirement for the Degree of Bachelor of Engineering

25 November 2005

## Acknowledgements

In working on this project I have benefitted from the generous help of many individuals. In particular Patrick L. Schmitz has allowed me to use his excellent design proposal which I have referred to extensively in this report and which has assisted my work tremendously.

Many in the Mozilla community have also very kindly given me guidance and assistance as I have tried to grasp and have wrestled with the Mozilla code. In particular I have valued the advice and assistance of Robert O'Callahan, Tim Rowley, Jonathan Watt and Alex Fritze.

Many thanks also to those who have followed my progress and provided feedback or encouragement. Zbigniew Braniecki's assistance in testing and Olaf Hoffmann's feedback and test suite have been particularly valuable.

*You are worthy, our Lord and God,  
to receive glory and honour and power,  
for You created all things,  
and by Your will they were created  
and have their being.*

— *Revelation 4:11*

## Implementing Scalable Vector Graphics in Mozilla

Brian Birtles

Spring 2005

As the World Wide Web has grown in popularity and sophistication many have realised its potential as more than just an information medium but as a platform for developing and distributing software applications. Applications built on this Web platform are often faster to develop, easier to deploy and maintain, and allow users greater physical freedom as Web applications can be used from any electronic device with an Internet connection and a Web browser. The common XML underpinning of the Web platform also opens up great potential for integrating and syndicating disparate information sources and technologies. So far however, applications built on the Web platform cannot provide the same rich user experience as their desktop counterparts and this prevents many solutions from being realised as Web applications.

This Capstone project contributes to the implementation of a graphics standard called Scalable Vector Graphics, or SVG, for a popular Web application platform, Mozilla. Such an implementation allows application developers to produce richer, more interactive Web applications much like traditional desktop applications.

Of the SVG features yet to be implemented in the Mozilla platform, declarative animation is one of the most significant and is frequently requested. A design is presented that attempts to address the full range of features defined for SVG's declarative animation and that is suited to the requirements particular to the Mozilla codebase. Using this design, the bulk of the animation model is implemented and is detailed in this report. It is anticipated that this implementation will form the basis for the remainder of Mozilla's SVG animation features and could potentially be extended to animate content other than SVG providing an even richer application platform still.

# Contents

<b>1</b>	<b>Background</b>	<b>1</b>
1.1	The Web platform to date . . . . .	2
1.2	Graphics on the Web . . . . .	4
1.3	SVG: Vector graphics for the Web platform . . . . .	6
1.4	SVG and Flash . . . . .	7
1.4.1	SVG: The XML-based standard . . . . .	7
1.4.2	SVG: The open standard . . . . .	8
1.5	Other alternatives to SVG . . . . .	10
1.6	State of the SVG-enabled platform . . . . .	11
1.7	Mozilla . . . . .	12
<b>2</b>	<b>Project outline</b>	<b>16</b>
2.1	Declarative animation . . . . .	16
2.1.1	Features of SMIL Animation . . . . .	20
2.2	Declarative animation as a university project . . . . .	22

<b>3</b>	<b>Engineering process</b>	<b>25</b>
3.1	Process influences . . . . .	25
3.2	The agile approach . . . . .	26
3.3	Open-source development . . . . .	27
3.4	Process overview . . . . .	28
3.5	Project familiarisation: Tackling a “first bug” . . . . .	31
3.6	The first prototype . . . . .	32
3.7	The second prototype . . . . .	32
3.8	Requirements analysis . . . . .	33
3.9	Design strategy . . . . .	34
3.10	Test strategy . . . . .	35
3.10.1	Unit tests . . . . .	35
3.10.2	SVG test cases . . . . .	36
3.10.3	Public test suites . . . . .	37
3.11	Communication . . . . .	38
3.11.1	Documentation . . . . .	39
3.11.2	Risk management . . . . .	40
<b>4</b>	<b>Analysis</b>	<b>42</b>
4.1	Analysis of the SMIL model . . . . .	42
4.1.1	Data-type agnostic animation . . . . .	42
4.1.2	Additive composition . . . . .	44
4.1.3	Timer independence . . . . .	44

4.1.4	Dynamic document model . . . . .	45
4.1.5	Dynamic dependencies . . . . .	45
4.2	Analysis of the Mozilla codebase . . . . .	46
4.2.1	Netscape Portable Runtime . . . . .	46
4.2.2	XPCOM . . . . .	47
4.2.3	Utility of SMIL . . . . .	48
4.2.4	The creation of a platform . . . . .	48
4.2.5	Contributing to a mature product . . . . .	49
<b>5</b>	<b>Design</b>	<b>50</b>
5.1	Design process . . . . .	50
5.2	Operation overview . . . . .	54
5.2.1	Document setup . . . . .	54
5.2.2	Sampling . . . . .	54
5.2.3	Composition . . . . .	55
5.3	Comparison with Schmitz's model . . . . .	57
5.4	Timing model . . . . .	57
5.4.1	Animation controller . . . . .	57
5.4.2	Timed document root . . . . .	59
5.4.3	Timed element . . . . .	59
5.4.4	Time client interface . . . . .	60
5.4.5	Time value specification . . . . .	60
5.4.6	Interval . . . . .	60

5.4.7	Instance time . . . . .	60
5.4.8	Time value . . . . .	61
5.5	Animation model . . . . .	61
5.5.1	Animation registry . . . . .	63
5.5.2	Animation observer . . . . .	64
5.5.3	Compositor . . . . .	65
5.5.4	Animation function . . . . .	65
5.5.5	Animated value interface . . . . .	66
5.5.6	Animated attribute interface . . . . .	66
5.5.7	Animated element interface . . . . .	66
5.6	Design evaluation . . . . .	67
<b>6</b>	<b>Implementation details</b>	<b>69</b>
6.1	Controlling the animation start-point . . . . .	69
6.2	Integrating SVG data types . . . . .	71
6.3	Frozen ‘to animation’ . . . . .	72
6.4	Spline-based timing . . . . .	77
6.5	Ownership . . . . .	82
6.6	Performance tuning . . . . .	83
6.7	Threading . . . . .	84
<b>7</b>	<b>Conclusions</b>	<b>85</b>
7.1	Implementation status . . . . .	85
7.2	Future directions . . . . .	88



<i>CONTENTS</i>	viii
<b>Glossary</b>	<b>92</b>
<b>References</b>	<b>95</b>
<b>A Requirements database</b>	<b>101</b>
<b>B Test results</b>	<b>107</b>
B.1 Animation function tests . . . . .	107
B.1.1 TestSampleAt . . . . .	107
B.1.2 TestParameterPriority . . . . .	108
B.1.3 TestValuesParsing . . . . .	109
B.1.4 TestAdditive . . . . .	109
B.1.5 TestKeyTimes . . . . .	110
B.1.6 TestKeySplines . . . . .	113
B.1.7 TestAccumulate . . . . .	115
B.1.8 TestByAnimation . . . . .	116
B.2 Timed element tests . . . . .	117
B.2.1 TestOffsetStartup . . . . .	117
B.2.2 TestMultipleBegins . . . . .	117
B.2.3 TestNegativeTimes . . . . .	118
B.2.4 TestSorting . . . . .	118
B.2.5 TestZeroDurationIntervals . . . . .	119
B.2.6 TestMoreZeroDurationIntervals . . . . .	120
B.2.7 TestEndSpecs . . . . .	122

B.2.8	TestBlankBegin . . . . .	122
B.2.9	TestIndefiniteBegin . . . . .	123
B.2.10	TestBadInput . . . . .	123
B.2.11	TestUnsetting . . . . .	123
B.2.12	TestRepeatSpecs . . . . .	124
B.2.13	TestRepeating . . . . .	125
B.2.14	TestFillMode . . . . .	126
B.2.15	TestRestartMode . . . . .	128
B.2.16	TestMin . . . . .	129
B.2.17	TestMax . . . . .	131
B.2.18	TestMinAndMax . . . . .	133

# List of Figures

1.1	Comparison between raster and vector graphics. . . . .	5
2.1	A simple SVG document using DOM-based animation. . . . .	17
2.2	A simple SVG document using declarative animation. . . . .	18
2.3	A possible work-flow where XSLT is used to transform an SVG document animated with SMIL into a static timeline of the same animation, drawn using SVG. . . . .	19
3.1	The Agile Manifesto. . . . .	26
3.2	The initially envisioned spiral process. . . . .	30
4.1	A set of animation elements targeting different attributes on the same target element. . . . .	43
4.2	An example of two animations targeting the same attribute. . . .	44
5.1	Class diagram of the design prior to incorporating Schmitz's ideas.	51
5.2	Overview class diagram after incorporating Schmitz's ideas. . . . .	53
5.3	Sequence diagram for a single animation sample. . . . .	56
5.4	Class diagram of the timing model. . . . .	58
5.5	Class diagram of the animation model. . . . .	62

6.1	A simple animation that causes a circle to follow a triangle-wave pattern. . . . .	73
6.2	An animation composed of the triangle wave from Figure 6.1 combined with a ‘to animation’. . . . .	73
6.3	Alternative behaviour for a frozen ‘to animation’. . . . .	76
6.4	The spline graph used by the <code>keySplines</code> attribute. . . . .	78
6.5	Screenshot from the <code>keySplines</code> test graphic. . . . .	82

# List of Tables

7.1 SMIL Animation features implemented in this project. . . . . 86

A.1 Requirements database . . . . . 101

# Chapter 1

## Background

“Everything in software changes. The requirements change. The design changes. The business changes. The technology changes. The team changes. The team members change.” — Kent Beck

Having seen more than its fair share of silver bullets and breakthrough paradigms, the high-technology industry, and particularly the software industry, has started to regard “the next big thing” with more cool indifference than zealous enthusiasm. It is often said that in such an industry, change is the only constant. Recognition of this fact is the primary motivation behind Beck’s (2000) Extreme Programming, the motto of which is “Embrace Change”.

At the same time, it is usually not hard to see that many of these new trends are in fact not new but are simply extensions of older developments (Pike 2005), sometimes much older. For example, XML, one of the most popular developments of recent times is derived from SGML, developed in the 1960s. The context and the applications of these technologies however are new.

In this chapter and the next we survey some of the emerging technologies in the field of Web application development. However, as we shall see, many of these technologies are derived from established and proven concepts but adapted to the context of the Web, they provide new possibilities.

This chapter describes the context in which some of these technologies have emerged and therefore the context for my Capstone project. The next chapter describes the technologies that my project deals with from a technical point of view.

## 1.1 The Web platform to date

When the World Wide Web was first developed in 1989, it was designed to combine “the techniques of information retrieval and hypertext to make an easy but powerful global information system” (Berners-Lee 1990). While this must have seemed like an ambitious goal in 1989, it might now almost be considered too modest. Since the first Web page was produced in 1990 the popularity of the Web has grown exponentially (Bell 1999, p. 9).

The popularity of the Web led in turn to the omnipresence of the Web browser. It is now the case, that any computer, anywhere, running any operating system will almost certainly be equipped with a Web browser, and the same will soon be true for mobile phones, televisions and even refrigerators.

For a long time many in the software industry have regarded the idea of being able to “write once, run anywhere” as a sort of holy grail (Torvalds 1999, p. 101). This was the primary goal of the Java platform but political and technical issues have so far prevented it from reaching its full potential as a client-side technology (Bucken 2003). The Web, however, offers an even more attractive platform for application development. Not only is it available on nearly every computer and perhaps some day on nearly every electrical appliance—something the Java platform nearly achieved—but Web applications are hosted by a central server which allows easy maintenance, requires no installation effort on the user’s part, and, perhaps most importantly, allows the same data to be accessed regardless of the user’s physical location.

This is not, however, a new idea. The idea of hosting applications on a central computer has existed since the first mainframe computers, distributed applications have existed since at least ARPANET in 1969, and the notion of allowing

users more physical flexibility in their use of computers is just one of many concepts that have long been discussed under the name “ubiquitous computing”. The novelty of the Web is simply that it is available now, it is an open platform as we shall discuss later, and it is approaching omnipresence.

However, the story of the Web up until now has not been entirely smooth. Along with its growth in popularity, greater demands have been made of the Web. The Web was originally intended to convey information independent of any particular presentation, but with the rise in the Web’s popularity and broadening of its user base came demands for control over the formatting of Web pages. This demand was met, in some cases, by developing new standards such as Cascading Style Sheets (CSS), but most often was met by individual Web browser vendors extending Web standards in non-standard ways. This uncoordinated innovation left many of those developing content and software for this medium frustrated.

After a frenzy of competition in the late 1990s known as the browser wars, Microsoft Internet Explorer supplanted Netscape Communicator as the dominant Web browser. After this point the development of new client-side Web technologies stagnated (Granneman 2005, p. 38). Instead, Web developers, as they came to be known, focused on server-side technologies. This approach consists of performing the bulk of the computational work on a central server and then sending a snapshot of the information to a client, often using the lowest common denominator of technologies on the client-side. This trend has seen the rise of scripting languages such as PHP and Python combined with easy-to-use databases such as MySQL and PostgreSQL. Once again, scripting languages and databases are as ancient as Computing Science itself but are now applied in the context of the Web. More recently, a greater emphasis has been placed on connecting such systems using technologies such as Simple Object Access Protocol (SOAP); Web Services Description Language (WSDL); Universal Description, Discovery, and Integration (UDDI) and to some extent Really Simple Syndication (RSS) and XSL Transformations (XSLT).

All the while however, little has changed on the client-side. While initial descriptions of the Web were forward looking enough to consider ‘virtual documents’ (now called dynamically generated content—the focus of server-side technologies)



and multimedia, they were still very much steeped in the language of ‘documents’ and ‘requests’. As Garrett (2005) explains, this model “makes the Web good for hypertext [but] doesn’t necessarily make it good for software applications.”

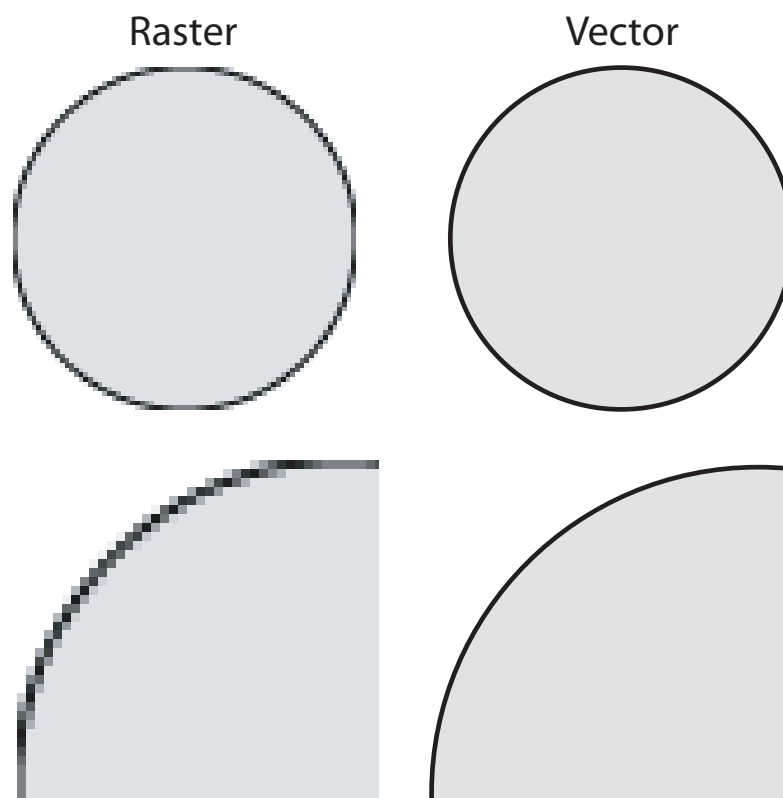
Combining the few recent developments in client-side Web technologies, Garrett explains how Asynchronous JavaScript and XML (AJAX) can be used to overcome some of the limitations of this model. Yet in his introduction Garrett claims that, “Desktop applications have a richness and responsiveness that has seemed out of reach on the Web.” While AJAX addresses the issue of responsiveness, the matter of richness remains.

## 1.2 Graphics on the Web

Much of the functionality on offer to desktop application programmers is simply not available to developers working with Web technologies such as HTML, JavaScript and CSS. This is especially true for those Web developers adhering to the lowest common denominator of support for standards provided by different Web browsers.

As an example, a Web developer does not even have access to basic drawing functions. In order to draw a circle, perhaps to highlight some information or to use as a button, a Web developer must draw the circle first in a graphics program; export it in an appropriate file format, size and colour depth; store it on a Web server; and then create a link to the file. The export step requires particular attention in order for the graphic to blend smoothly into the Web page and in fact is the subject of whole books. This entire process must be repeated to change the colour, shape or size of the circle. Furthermore, the graphic will generally produce poor results when printed or used on mobile devices.

Compare this to a desktop application where a circle is typically drawn using a function call to a graphics library, possibly provided by the operating system. The circle produced in this manner blends into its surroundings, appears smooth regardless of the destination and can often be recoloured quite effortlessly. Figure 1.1 demonstrates the difference in smoothness between the two techniques.



**Figure 1.1:** Comparison between the raster graphics typically used in Web applications (left) and the vector graphics primitives available to desktop applications (right) and a magnified section of each graphic below.

Of course, those familiar with computer graphics would realise that these differences are simply the differences between raster or bitmapped graphics and vector graphics. Whereas raster graphics describe images in terms of dots, vector graphics describe an image through mathematical descriptions of lines. As a result, raster graphics are generally better suited to describing photos, whereas vector graphics are better suited to describing line art but as we shall see, vector graphics have other properties that make them particularly suitable to application development.

### 1.3 SVG: Vector graphics for the Web platform

In September 2001, the World Wide Web Consortium (W3C) presented a new standard for describing two-dimensional graphics in XML called Scalable Vector Graphics (SVG). This format provides all the advantages of vector graphics we have already discussed and in that sense is not a new technology. The context, however, is different.

Not only does SVG bring vector graphics to the domain of Web applications, but because it is an open and XML-based format, SVG forms part of a growing set of standards which collectively are coming to be known as the Web *platform*.

Aulenback (2004) picks up on the importance of SVG as a component of the Web platform when he asserts that despite the promise of Web Services, “The current missing item in Web Services is the visual interface”. SVG meets this need by providing the sort of graphics needed for creating visual interfaces. Several features of SVG make it particularly suited to this situation.

Firstly, SVG graphics are scalable. This makes them suitable for use on a range of devices. For this reason, SVG has so far proved most successful on mobile devices where screen sizes vary considerably and making effective use of the limited screen real estate is particularly important. In July 2005, [svg.org](http://svg.org) reported that more than 50 mobile phones were capable of displaying SVG according to the SVG 1.1 Tiny Specification ([svg.org](http://svg.org) 2005).

Another important quality of SVG is that objects in an SVG graphic retain their individual identity allowing them to be manipulated via style sheets and scripts. Cagle (2004) emphasises this point, claiming that “the real advantage of vector graphics comes from the fact that any such graphic is semantically modular”. He then goes on to explain how this fact can be exploited to produce user interfaces for large-scale distributed applications.

Finally, as a further benefit, SVG graphics are often smaller in size and therefore faster to transfer between computers than their raster equivalents.

## 1.4 SVG and Flash

Anyone familiar with current technologies in use on the Web will quickly notice similarities between SVG and Macromedia Flash. A feature-for-feature comparison of the two technologies reveals a considerable degree of overlap (Neumann & Winter 2004). However, at least two very substantial differences lie beneath the surface.

### 1.4.1 SVG: The XML-based standard

The first difference is the underlying technology. Because SVG is an XML-based format it integrates seamlessly with other XML-based technologies including the bulk of Web standards. For example, XSL Transformations (XSLT) can be used to translate a Chemical Markup Language (CML) document into a visual representation described using SVG.

Furthermore SVG has been designed with integration with other specific Web standards in mind. For example,

- the ECMAScript scripting language also known as JavaScript can be used to manipulate SVG elements using the same Document Object Model as is used to manipulate an XHTML document;

- Synchronised Multimedia Integration Language (SMIL) can be used to animate both SVG and generic SMIL objects within a Multimedia Messaging Service (MMS) document;
- Ruby could be used to annotate Japanese text in an XHTML fragment within an SVG diagram;
- a user interface described with XML User Interface Language (XUL) might use SVG to describe the toolbar icons;
- and the same Cascading Style Sheets (CSS) specification could be used to style a combination of SVG and XHTML content.

The possibilities for combining these technologies are so broad that in August 2005 the World Wide Web Consortium (W3C) released the first draft of several documents describing such use cases and requirements for this kind of integration (World Wide Web Consortium 2005*a*, World Wide Web Consortium 2005*b*). Such scenarios are referred to as compound documents but this is not the only mode of operation supported by the Web platform.

Scripting languages such as ECMAScript, also known as JavaScript, can be used to manipulate the elements of an XML document using a standard interface known as the Document Object Model (DOM). This same interface can be used to manipulate an SVG diagram, an XHTML Web page, an X3D virtual world or all three. While it is also possible to manipulate objects in a Flash file using ActionScript this approach does not scale well when dynamic manipulation of other, XML-based standards are involved. As Aulenback (2004, p. 86) asserts, without “XML from end to end . . . the links in the chain are broken and the value proposition quickly lost”.

### 1.4.2 SVG: The open standard

The second difference that separates SVG from Flash is that SVG is an open standard. In his keynote presentation at the SVG Open 2005 Conference, Cagle provided the following definition of an open standard,

A standard is in essence an agreement to speak a common language. An open standard is the further agreement that anybody can speak that same language, that one didn't need to speak some form of secret "club-speak" known only to its members. (Cagle 2005)

In that sense, because SVG is an open standard, it is open to any party to produce tools, content or other services based on this format. To demonstrate this concept consider the following ways in which SVG is already used:

- Adobe Illustrator can load and save artwork in SVG format with little lost information;
- Ikivo Animator can be used to add animation effects to artwork, such as that produced by Adobe Illustrator;
- Microsoft Visio can save diagrams in SVG format;
- The Open Clip Art Library contains a catalogue of thousands of Public Domain clip art images, most of which are stored in SVG format;
- Apache Batik can be used to view SVG files and convert them into bitmapped representations.

In fact this list could go for several pages, particularly if we were to include the many open-source diagramming products that export SVG or use SVG as their native file format. The important point however is that in each case entirely independent vendors are producing the products that use SVG. This is a key advantage of using an open standard. Many have pointed out the benefits of openness as a risk management strategy to avoid vendor lock-in (Aulenback 2004, Bunker n.d.).

Another benefit of an open standard is that no one party has complete control over the development of the specification. This reduces the risk somewhat that the standard will be unexpectedly abandoned.

Some may argue that all this talk of openness is too idealistic; the sort of stuff touted by academics and bright-eyed teenagers but lost on the business world. However, we need look no further than the Web to see that this is not the case. Clearly much of the reason behind the success of the Web is that so many independent parties have been able to develop tools and content for the new medium and have been able to influence the direction of standards such as HTML. As a result of this growth, many new companies from tool vendors to search engines have emerged and prospered.

All this is not to discredit the Flash (SWF) file format. It has achieved great popularity because it meets an important need on the Web. However, for those developing for the Web platform with all its associated XML-based standards, SVG is a more natural fit. Its grounding in XML makes it more scalable and flexible in such solutions and its status as an open standard provides room for new tools and content to be developed by independent parties.

## 1.5 Other alternatives to SVG

So far we have discussed some of the current trends in the development of the Web as a platform for application development and we have seen how SVG attempts to fill an important gap in that platform. It should be noted that SVG is not the only technology seeking to address this gap. We have already considered Macromedia Flash but there are others too and at least one is worth mentioning here.

Disappointed with the direction of new Web standards, several developers of Web browsers formed a new standards body in 2004 named the Web Hypertext Application Technology Working Group (WHATWG). The charter of this group is to create technical specifications to “address the need for one coherent development environment for Web applications . . . in mass-market Web browsers” (WHATWG 2004). One product of this group is the Web Applications 1.0 spec-

ification (WHATWG 2005) which includes a canvas element for direct graphics programming.

The canvas element provided by the Web Applications specification allows a Web developer to issue graphics commands using a scripting language in much the same way a procedural graphics interface provided by a graphics library or operating system might work.

Although the programming interface for the canvas element has some similarities to SVG it has a different target usage to SVG. Because drawings on a canvas element are defined using a graphics commands inside a scripting language, they cannot be authored in the same fashion as SVG because an authoring program would need to be able to understand the scripting commands which may be interleaved with any combination of programming control structures. The canvas element is not intended to be used in this way however; rather, it is intended for producing “visual images on the fly” (WHATWG 2005) such as for games or graphs. For such situations, this mode of interaction and the much simpler feature set of the canvas element make it a suitable choice, but for development of Web applications, SVG is often a more suitable candidate.

## 1.6 State of the SVG-enabled platform

I hope that the purpose of SVG is now clear. What remains then is to see how this theory and speculation translates into practice. We have already seen several examples of SVG in use in real applications but the rich Web platform, enlivened by SVG is still yet to emerge.

Adobe have produced a plug-in to allow SVG to be viewed within a Web browser but this plug-in is not yet widely distributed enough to make SVG commonplace on the Web. Furthermore, as many have pointed out, there is a distinct disadvantage to implementing SVG as a plug-in component.

Rowley, Morris, Watt & Fritze (2005) note that “plugin content appears ‘in a box’ both in visual terms and from the point of view of the DOM”. That is to



say plug-ins typically do not integrate well with other graphical elements or with the Web browser's object model. This second point is particularly pertinent in the context of building a platform for application development. Plug-ins also present problems with regards to security, installation difficulties and platform availability (Cagle 2005, Rowley et al. 2005).

For this reason many parties are pinning their hopes for SVG on more integrated solutions. One example of this is Mozilla.

## 1.7 Mozilla

In 1998, Netscape Communications decided to release the source code to the Netscape Web browser and allow outside parties to contribute to its development. The name Mozilla, originally the codename for the Netscape Web browser and its Godzilla-like mascot, was used as the name for the collection of source code and also the project that surrounded its development.

Over time the proportion of contributors outside Netscape to those within grew until the Mozilla project could carry on in its own right, independently of Netscape (who had then been bought by AOL). In 2003, when AOL laid off several employees involved in developing Mozilla, the Mozilla Foundation was formed to oversee the future of the Web browser (Marson 2005) and in August 2005, the Mozilla Corporation was launched as a taxable subsidiary of the Mozilla Foundation (mozilla.org 2005b).

Since Netscape released the source code in 1998 it has come to be used not only for the Netscape Web browser but also for the Firefox and Camino Web browsers; the Thunderbird email client; the Mozilla Application Suite that integrates Web browsing, email and other functionality; and various other applications including several calendaring applications. For this reason, the term Mozilla is now often used to refer to the codebase from which several applications are drawn rather than a particular Web browser.

In considering the future of the Web platform, Mozilla is a particularly interesting case for a number of reasons.

Firstly, Mozilla is a well-established platform providing rigorously tested support for many Web standards including HTML and XHTML, CSS, DOM, RDF, ECMAScript, XSLT, Namespaces in XML, MathML, SOAP and XPath to name just a few.

In addition to the standards supported by Mozilla, the Mozilla project has produced several new technologies that are particularly relevant to the field of Web application development. Such technologies include the XML Binding Language (XBL) which has been submitted to the World Wide Web Consortium (W3C) for standardisation and the XML User Interface Language (XUL). XUL provides developers with the means to define user interfaces for their Web applications using similar components to those found in desktop applications.

Secondly, Mozilla is a well-established codebase designed to work with a broad range of platforms and compilers (Williams 1998). As a result, applications supported by Mozilla, can be run on nearly any computer.

Finally, and perhaps most importantly, the Mozilla-based Firefox Web browser is growing rapidly in popularity. It is hard to report exact figures, particularly as they will quite quickly be out of date, but it is certainly true that after Microsoft Internet Explorer, Mozilla Firefox is the second most popular Web browser in use today.

All of these factors make Mozilla a very attractive platform for Web application development, and a platform that would be even more attractive were it to support SVG. For this reason, since 1999 a small number of developers have been seeking to bring SVG to Mozilla. With the anticipated release of Firefox 1.5 in late 2005, SVG will finally arrive in Mozilla. However, this will not be a full implementation of the entire 700-page SVG 1.1 specification, but rather something like “SVG Full 1.1 minus filters, declarative animation, and SVG defined fonts” (Rowley et al. 2005). Of course, a full implementation is planned, perhaps some time into 2006 (Rowley et al. 2005). It is the goal of my Capstone project, to contribute to that effort.

Before concluding this chapter it is necessary to describe one final trend in software development in order to provide a more complete picture of the technological environment of this project. Since at least 2004, the Mozilla project has been developing a runtime environment for more traditional desktop applications that make use of many of the components developed as part of the Mozilla codebase. This runtime environment, known as XULRunner, will eventually form the platform upon which Mozilla's current products such as Firefox and Thunderbird are based. However, this platform could conceivably be used to develop all manner of desktop applications. In practice however, such applications will typically be those that will benefit from the many Web-related components available in the Mozilla codebase and its cross-platform foundations.

Some have compared XULRunner to Microsoft's .NET Framework (McFarlane 2003a) whilst others have been quick to point out the differences in focus of the two platforms (Markham 2005, Deakin 2003). If such a comparison is permitted however then SVG's Microsoft-based counterpart is the Extensible Application Markup Language (XAML). Indeed, many have compared these technologies but it is not useful for us to do so now and I will return to this topic only at the conclusion of this report.

Until now when I have referred to the Web platform and Web applications I have had in mind those applications that run entirely inside a Web browser and potentially any Web browser at that, rather than being restricted to a Mozilla-based browser. Quite clearly, XULRunner is directed toward another kind of application. Therefore I refer to the platform provided by XULRunner as the Mozilla platform rather than the Web platform. The Mozilla platform addresses an important gap but my focus in this project is in strengthening the Web platform.

In a preface to a collection of articles discussing SVG titled "Correcting the great mistake" Parisi (2004) emphasises that the purpose of technologies such as SVG is to communicate *information*. However in this chapter I have focused on the potential of SVG to be used as part of the Web platform, that is, for application development. There are other uses for SVG however. It may open

up new workflows in a desktop publishing environment or it may simply be used to display visual information on the Web in a more natural manner. However, to some degree such scenarios are already possible with existing technologies. SVG may be a better or more uniform way of achieving these results, but if SVG actually provides anything new to the fields of computer graphics and software development then I believe it is the ability to integrate graphic elements as first-class citizens amongst a host of other technologies used to develop applications for the Web. By contributing to the development of SVG in a popular Web browser this Capstone project attempts to further the realisation of a richer platform for Web application development.

# Chapter 2

## Project outline

Of all the features missing from Mozilla’s SVG codebase at the time when I first began this project, three were continually raised as the big-ticket items. These were declarative animation, filters and SVG-defined fonts. Of these, I have chosen to implement declarative animation for reasons that I will explain later in this chapter but before I do so, it is necessary to understand what declarative animation is and why it was included as part of the SVG specification.

### 2.1 Declarative animation

The name “declarative animation” suggests a particular type of animation, and this is because it is possible to produce animation effects in SVG using two distinct techniques.

The first animation technique involves manipulating SVG objects over time using a scripting language to access the SVG objects via the SVG DOM—an API for SVG documents. This is similar to the way in which animation is performed in many desktop applications. This mode of animation is often referred to as scripted or DOM-based animation.

The second technique by which SVG documents may be animated requires that the parameters of animation be described as part of the SVG document, that is,

they are “declared” as part of the document. The model and syntax used for declaring such animations in SVG belong to another standard, Synchronised Multimedia Interaction Language (SMIL—pronounced “smile”) Animation (World Wide Web Consortium 2001). For this reason this type of animation is often referred to as declarative or SMIL animation.

To appreciate the difference between these two modes of animation, consider a simple scenario where a circle is to be moved to the right over two seconds. The difference between these two modes is shown in Figures 2.1 and 2.2.

```
<?xml version="1.0" standalone="no" ?>
<svg xmlns="http://www.w3.org/2000/svg" width="200px" height="100px"
  onload="animate()">
  <script type="text/ecmascript">
    <![CDATA[
      function animate()
      {
        var circle = document.getElementById("target");
        circle.cx.baseVal.value++;
        if (circle.cx.baseVal.value < 150)
          setTimeout("animate()", 20);
      }
    ]]>
  </script>
  <circle id="target" fill="red" stroke="black" cx="50" cy="50" r="40"/>
</svg>
```

**Figure 2.1:** A simple SVG document using DOM-based animation.

There are advantages of using each approach. For the first case, animation using the SVG DOM, the author is not limited to a particular set of animation features but can achieve any effect possible using the scripting language and the SVG DOM. Some Web authors are already familiar with this style of interaction and so they do not need to learn any new techniques to provide animation using this method.

There are, however, many disadvantages to this approach. First of all, authors are

```
<?xml version="1.0" standalone="no" ?>
<svg xmlns="http://www.w3.org/2000/svg" width="200px" height="100px">
  <circle fill="red" stroke="black" cx="50" cy="50" r="40">
    <animate attributeName="cx" from="50" to="150" begin="0s" dur="2s"/>
  </circle>
</svg>
```

**Figure 2.2:** A simple SVG document using declarative animation.

forced to solve the same problem over and over again, namely, that of writing an animation loop. For simple animations this is trivial although the code snippet provided in Figure 2.1 is a poor example as it fails to account for time taken within the loop itself.<sup>1</sup>

A second disadvantage of the scripted approach is that it requires programming code to be written. This limits the ability of non-programmers to create animations but not entirely as some tools for Web development are able to produce small scripts to address common problems.

The most significant shortcoming of animating with script and the SVG DOM is that such animations are opaque to the computer. Without actually running the script, the computer will not be able to determine what it does. Scripted animations cannot be manipulated using an authoring application in any kind of intuitive manner.<sup>2</sup> King, Schmitz & Thompson (2004) add, “the use of script or code is problematic in data-driven content models based upon XML and associated tools”. As an example, it would be very difficult to produce an animation script from some information in an XML data store by using XSLT.

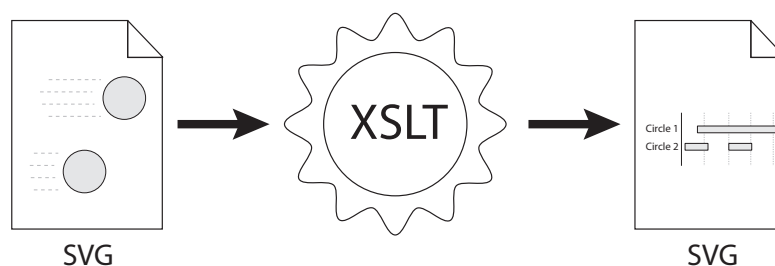
Declarative animation seeks to address these problems. As shown in Figure 2.2 the parameters of declarative animation are laid bare as regular XML attributes. This enables such animations to be edited by any number of applications so that

---

<sup>1</sup>For a slightly more sophisticated example, see section 19.3 of the SVG 1.1 Specification.

<sup>2</sup>Although this is generally true, it may be possible for an application to use careful variable naming, or special comments so that it is able to re-interpret the script it has produced but such an approach is essentially a primitive form of declarative animation anyway and the animation produced will not be able to be edited by other applications.

users are not tied to a particular application or a particular work-flow. Because any application can understand the animation, alternative representations are possible. For example, it may be possible to produce an application that draws a timeline of the animations defined for a particular SVG graphic. In fact such a timeline could even be drawn in SVG using an XSLT transformation as shown in Figure 2.3.



**Figure 2.3:** A possible work-flow where XSLT is used to transform an SVG document animated with SMIL into a static timeline of the same animation, drawn using SVG.

Declarative animation provides a set of common animation features such as interpolating values and coordinating otherwise independent transitions. Provided that these features correspond to a Web author’s requirements for an animation—and standards such as SMIL Animation make every effort to ensure this is the case—then it is also likely to be significantly faster and easier to produce animations using declarative syntax rather than script.

The standards body responsible for most of the standards in use on the Web today including SVG is the World Wide Web Consortium (W3C). The grand vision of this organisation’s founder, Tim Berners-Lee, is the Semantic Web (Berners-Lee, Hendler & Lassila 2001). This vision is based on giving *meaning* to the data that exists on the Web so that computers may act on the data and combine it more intelligently. Scripted animation hides the semantics of the animation from the computer, whereas declarative animation makes the animation parameters visible to the computer for the purpose of creating new animations from other data, manipulating the animation or providing alternative representations of the animation.



We have already mentioned one obvious disadvantage of declarative animation: the author is limited to the features provided by the animation description language. Although SMIL Animation provides a very large feature set to content authors there will still be cases where these features are not sufficient and scripting is required. King et al. (2004) acknowledge this situation and propose a sort of middle-ground combining the two approaches by using functional programming together with SMIL Animation.

The advantages of declarative markup versus scripting are still under discussion. For example, prior to a workshop on Web Applications and Compound Documents the World Wide Web Consortium (2004) posed the question, “How much of a Web application should be declarative? How much in script?”.

It should at least be clear that even if a declarative approach is not always the most appropriate choice it is not redundant either. Because SMIL animation offers Web developers the benefits I have described it is important that it be implemented in Mozilla. Furthermore, even the most limited subset of the SVG specification defined by the W3C, SVG 1.1 Tiny, includes declarative animation. Therefore, in order for Mozilla’s SVG implementation to be able to claim conformance to any SVG profile, declarative animation must be implemented.

### 2.1.1 Features of SMIL Animation

SMIL Animation is the name of the standard upon which SVG’s animation constructs are based. This standard defines the syntax and semantics of these elements and is a subset of the SMIL 2.0 specification.

Schmitz (n.d.) describes how this standard can be analysed as consisting of two separate models.

The first model is the timing model. This model includes all the features relating to defining and triggering animation intervals and sampling during these intervals. Some of these features include:

- The ability to define animations as starting relative to when a document is loaded or to a real clock value such as 10pm on 5<sup>th</sup> February 2006.
- The ability to define animations as starting relative to one another. For example, element A might be defined to start five seconds after element B. This is often referred to as synchbase timing.
- The ability to define animations to start relative to events such as keyboard presses, mouse clicks or events generated by programming scripts.
- The ability to define multiple intervals for an animation such that it can be played several times based on different conditions.
- The ability to constrain animation intervals. For example, an animation may be defined to stop when a mouse is clicked, when another animation starts or after some maximum amount of time has expired.
- The ability to specify if and how an animation should repeat.
- The ability to seek to a point in the animation using hyperlinks.
- The ability to define if an animation continues to affect its target after it has completed or if the target is reset.
- The ability to manipulate animations using DOM interfaces and receive feedback on animation status using DOM events.

The second model we can consider in approaching SMIL Animation is the animation model. Whereas the timing model deals with intervals, times and sampling, the animation model deals with the actual animation values. This includes interpolating between animation values, combining values for overlapping animations and applying the animated values to the target. The features of this model include:

- The ability to specify how interpolation is performed between key animation values. For example, the values could be interpolated in a linear fashion over time, or by using a function that creates an ease-in, ease-out effect.

- The ability to define animation values using one of several notations. The animation may be defined as proceeding from one value to another or as following a list of values or as converging on some final result independent of the intermediate values.
- The ability to adjust the time at which values are set so that they are not required to be evenly spaced.
- The ability to combine independent animations targeting the same property. The priority of each animation is determined and then each animation can define if it adds to lower priority animations or replaces them.
- The ability to define if repeated animations build upon the previous iteration or start afresh.
- The ability to animate an object along a path.
- The ability to animate between colours.

In addition to these animation model features, SVG adds an additional element to provide support for animating transformations so that any object can be translated, scaled, rotated, and skewed over time.

Compared to SMIL 2.0, the most notable difference is that SMIL Animation lacks time containers which allow timing elements to be grouped and then synchronised. The timing model used in SMIL Animation consists of only a single time container.

I will discuss some of these features in more detail later but for now it is sufficient to be aware of the possibilities provided by SMIL Animation.

## 2.2 Declarative animation as a university project

I have described why I consider declarative animation to be important to Mozilla. For these reasons I have chosen to implement this feature for my Capstone

project. Declarative animation also makes a suitable project because parts of it entail the sort of algorithmic complexity that is difficult to develop effectively in a distributed manner.

My purpose in conducting this project is to make a valuable contribution to the ongoing effort to support for SVG in Mozilla. Although animation is only one chapter amongst many in the SVG 1.1 specification much of the semantics of the model described in this chapter are properly defined in the separate SMIL Animation specification. The animation features of the SMIL Animation specification are not trivial and a complete implementation of these features is well beyond a Capstone project. The scope of this project is therefore limited to a subset of the animation features of SMIL Animation.

In determining the subset of SMIL Animation to implement I have chosen to focus my efforts toward developing a useful architecture that is suitably integrated with the existing codebase such that it could be accepted into the Mozilla project.

My reason for focusing on a useful architecture is to assist other developers. I have mentioned that SMIL Animation is not trivial and having reviewed this specification I would like to embody my understanding by developing and implementing a suitable architecture. My hope is that this will allow other developers to implement specific features without requiring them to be familiar with the SMIL Animation specification in its entirety.

In implementing this project I will be aiming to produce code such that it could be integrated into the existing codebase. It is quite a different matter to implement a feature and to integrate that feature, especially when the target is a codebase as complex as Mozilla.

Integrating in this case includes re-using already existing services, creating the necessary hooks in existing components, producing objects in the appropriate scope, modifying existing objects to inter-operate with the new functionality and factoring in the performance and threading requirements of the target. This work also includes discussing implementation strategies with interested parties and producing appropriate documentation such that other developers can continue the work.

Please be aware that in my original Capstone project proposal I emphasised that having my code accepted into the Mozilla codebase would not be a requirement of this project. This is because the approval process is beyond my control and can take many months, particularly for a large feature such as I am implementing and I cannot expect this process to be expedited for the sake of this project. Therefore, although I have every expectation that the code I produce will at some stage become part of the Mozilla codebase, this is not expected to happen prior to the submission of this project.

# Chapter 3

## Engineering process

“Each problem that I solved became a rule which served afterwards to solve other problems.” — Rene Descartes (1596–1650), *Discours de la Methode*

Over the span of this project I have had sufficient opportunity to consider and refine the processes that aided my work. In this chapter I will comment on some of these processes. I will also refer to the processes used by the Mozilla project to which I have been contributing and more generally, the processes common to many open-source projects. I hope that these comments will broaden the value of this discussion.

### 3.1 Process influences

Before describing the particular processes I have selected for this project I would like to comment on the major sources from which they are drawn. Without identifying these influences the processes I describe might otherwise seem to be a mélange of buzzwords and whimsical ideas rather than a critical distillation of established practice.

The two sources that have most influenced my approach to this project are agile software development and open-source software development, which, as we shall see are quite complementary in their values.

## 3.2 The agile approach

In recent years the intersection of several emerging software development processes such as Extreme Programming and Feature-Driven Development have come to be collectively referred to as agile methodologies. The common features and values of these approaches are captured in the Agile Manifesto (Figure 3.1).

We are uncovering better ways of developing software by doing it and helping others do it.  
Through this work we have come to value:

Individuals and interactions over processes and tools  
Working software over comprehensive documentation  
Customer collaboration over contract negotiation  
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

*Source:* Beck, Beedle, van Bennekum, Cockburn, Cunningham, Fowler, Grenning, Highsmith, Hunt, Jeffries, Kern, Marick, Martin, Mellor, Schwaber, Sutherland & Thomas (2001)

**Figure 3.1:** The Agile Manifesto.

The most appealing characteristic of this approach for my project is the value placed on responding to change. When uncertainty is high, as it is for this project, change is inevitable. The agile methodologies each provide different answers to the problem of change but in general each describes lightweight processes that

can be easily adapted to new requirements, invalidated assumptions or an altered environment. As a result I have borrowed many of my processes from this school of thought, but particularly Beck's (2000) Extreme Programming.

### 3.3 Open-source development

In contributing to a project such as Mozilla it is helpful to understand the culture surrounding the project. For Mozilla the culture is quite the same as for any open-source project. Himanen, Torvalds & Castells (2002) describe this culture at length under the title "The Hacker Ethic". In their analysis, the culture in such projects resembles more closely the academic ethos than the values common in a corporate environment. For example, open-source projects typically feature a high degree of transparency. This is true not only with regards to the program source code which gives this sort of development its name, but also with regards to design discussion, defect tracking and status reporting. One example of this is that anyone can view nearly any defect filed for the Mozilla project but the same cannot be said of its closed-source counterparts.

Himanen et al. (2002) compare the transparency of the open-source culture to the academic tradition of sharing the results of one's research for others to scrutinise and use. Further similarities can be drawn between these two cultures. For example, in each case the contributors are generally motivated by a genuine interest in their work and a desire to attain the regard of their peers rather than by financial incentives. Consequently in both circles scholarship is highly regarded.

This open-source culture has been another influential factor in the processes I have adopted for this project. For example I have deliberately tried to ensure transparency in my status reporting and design work.

In my opinion agile software development is well suited to the open-source culture. Open-source projects typically involve physically disparate contributors following different work patterns in different timezones. In such an environment it would



be difficult to enforce the sort of formality required to follow, for example, a process based on the waterfall model but it is much easier to manage using a lightweight process that is adaptable to change.

There is also evidence of the complementary relationship between agile methods and open source in the number of open-source products supporting these methods. For example, what is arguably the cornerstone of agile development in Java, JUnit, is an open-source product. Likewise, the Eclipse IDE which contains perhaps the best support of any IDE for refactoring—a technique common in agile methods—is open source as well.

This complementary relationship is not surprising. Both hackerism—Himanen et al.'s (2002) term for the dominant culture of the open-source world—and agile methods both put the emphasis of software development back on programming.

### 3.4 Process overview

In selecting a software process there can be many contributing factors to consider such as the volatility of the requirements, the abilities of the project team, the availability of representative users, and the characteristics of the project itself (Futrell, Shafer & Shafer 2002, pp. 146–152). For this particular project however the dominating factor I have used in selecting a process is risk management.

The project I have undertaken involved an incredibly large degree of uncertainty. At the outset I had only the faintest notion of the technologies involved and while it is normal to learn new technologies throughout a project the sheer number and complexity of technologies involved in this particular project made it critical to take an approach that would allow me to manage these unknown quantities.

Based on this consideration I elected to follow an iterative approach to development. By doing so I hoped to guard against the risk of being unable to deliver a demonstrable product at the completion of the project. Using an iterative approach I could be sure that by the completion of the first iteration I would either

have a demonstrable but limited product, or the realisation that the project was infeasible and sufficient time to plan an alternative course.

All agile methods follow an iterative approach of some sort. In Feature-Driven Development iterations are based around the features to be delivered. In Extreme Programming stories and tasks form the basis of iterations. In these cases the purpose of an iterative approach is to deliver value to the customer as quickly as possible and receive feedback to incorporate into the next iteration. My purpose for adopting an iterative approach however was largely a risk management measure although it is also true that I was able to receive early feedback on my work which I incorporated into subsequent revisions.

This approach of managing risk through iteration is most similar to Spiral Development as defined by Boehm (2000). Under this process model risk analysis is the first step in each iteration and determines the effort and detail required for each activity in the iteration and even the activities themselves. For this reason Spiral Development is defined as a process model *generator* because the activities within each iteration are not determined in advance. It is a sort of “meta-process”.

Based on these ideas I developed the approach shown in Figure 3.2. However, in keeping with Spiral Development I re-assessed the risks at each iteration and adjusted the activities of the iteration accordingly. For example, for reasons I will explain shortly, after the first iteration I was more confident of my design than I anticipated to be and scarcely performed any design work in subsequent iterations.

With regard to the process outlined in Figure 3.2 the requirements activities may seem surprisingly under-represented. This is deliberate. The requirements for this project entail little risk as they are well-defined in two formal specifications, namely SVG 1.1 and SMIL Animation. They are complex but there is little risk that they will change throughout the course of the project. Therefore it is sufficient to study the requirements once up front rather than revising them each iteration. However, I have scheduled the two prototype iterations prior to requirements analysis so that I can better estimate the amount of effort required

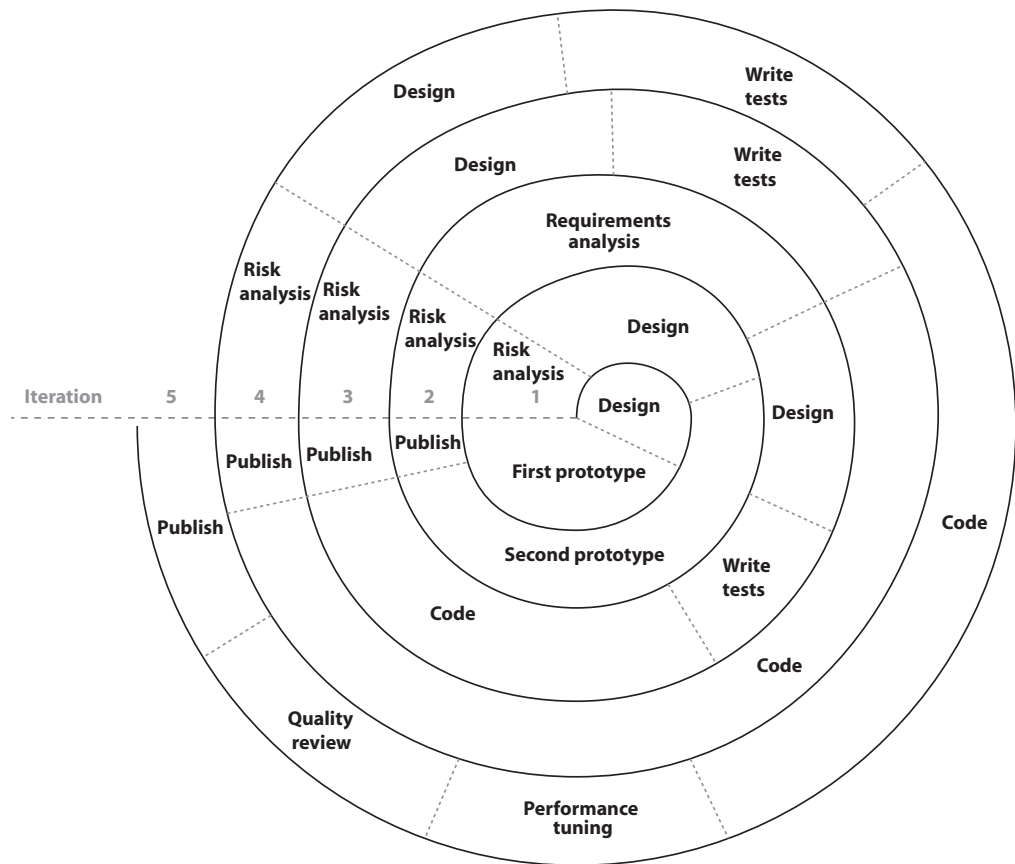


Figure 3.2: The initially envisioned spiral process.

by each requirement during analysis and planning for each iteration.

The ‘Publish’ activity at the end of all but the first iteration refers to the process of posting the results of my work so that others could view my progress. This step allowed me to receive feedback on my work, forced me to keep the product and code in a demonstrable state and by being transparent in this manner I hoped to ensure the work I was doing was not being duplicated by another contributor.

One minor discrepancy between my process and traditional Spiral Development is that I have replaced the formal anchor milestones defined by Boehm (2000) with milestones established with my supervisor as this seemed to be more appropriate for a project of this scope.

With this iterative approach based on Spiral Development I developed specific processes for each of the project activities. The remainder of this chapter outlines some of these particular activities.

### 3.5 Project familiarisation: Tackling a “first bug”

Having been warned that I would require at least six months to learn XPCOM—one of the underlying technologies used in the Mozilla codebase—and that I should find out what was involved before committing to such a project, my first step, even prior to developing the process I have just outlined, was to select a simple bug in the Mozilla bug database that I could use to learn more about the codebase and the feasibility of this project. This is now the recommended approach for new contributors and a list of suitable “first bugs” has been compiled (mozilla.org 2005a).

The bug I selected related to compositing SVG graphics against the background Web page (bug #134708). Fixing this bug, although straightforward, required several weeks of investigation and introduced me to some of the SVG architecture such as the different rendering back-ends involved. Taking this step was also useful in becoming familiar with the individual contributors involved in the project and the processes used.

## 3.6 The first prototype

Even after submitting my first patch I did not have sufficient understanding of the problem to be able to accurately plan the project or comprehend the requirements captured in the relevant specifications. To further reduce the risk involved in this project I employed two iterations of throw-away prototyping.

For the first prototype I set the target of moving a box across the screen regardless of how many design rules I had to break or nasty tricks I had to employ in the process. This is perhaps somewhat similar to the motto quoted by Beck (2000, p. 134) “Make it run, make it right, make it fast.” Here the emphasis was on reducing risk by proving that each of the steps involved is possible and developing a better understanding of the problem domain. ‘Making it right’ would be attempted later when I could better comprehend the problem.

In this first prototype all animation parameters were hard-coded and could not be easily altered to produce different animations. Nevertheless it proved to me that that it would at least be possible to produce some minimal animation and helped me to understand some of the more uncertain parts of the design such as managing the animation timer and initiating redrawing.

## 3.7 The second prototype

Following on from the first prototype I listed the limitations of the code I had written and identified the areas of uncertainty to address in the next prototype. For this second prototype I focused on producing a model that was more generic, removing many of the hard-coded parameters and replacing them with some primitive parsing support and generic handling. In producing this prototype I was forced to consider ownership and lifetime issues more closely. I also took one of the suggestions from the Mozilla SVG developer community regarding storage of animated values and included this in my prototype.

During development of the second prototype I frequently encountered situations where after a long compile the animation would fail to run at all. After lengthy

debugging sessions and several re-compiles I would correct the problem and animations would run again. Sometimes however this process would take an entire week. These problems would typically occur because I had failed to understand some part of the Mozilla codebase. In order to reduce the amount of time lost in this way I changed my approach in favour of making more incremental changes between successive compiles and committed to keeping the prototype in a state that is always demonstrable. Achieving this often required planning each change such that every step could be tested.

Fowler's (1999) work on refactoring is based on such an idea. Fowler breaks down code restructuring into a catalogue of common changes and for each one describes the steps involved. After applying such a set of steps, the structure of the program may be improved whilst leaving its behaviour unchanged. However, Fowler does more than just list common steps, he also gives valuable practical advice on when to apply these changes and argues for greater tool support in this area, something which I expect could provide significant benefits to programmers. This has been taken up in some areas such as the Eclipse IDE's refactoring support for Java. However, other languages are less accommodating of automated refactoring support. For example, the C++ preprocessor provides many possibilities for confusing such tools, particularly in projects as complex as Mozilla.

This formal approach to modifying the source code such that after each collection of steps the product is testable and demonstrable was the basis for my approach in the second prototype.

### **3.8 Requirements analysis**

Following on from prototype development I began a detailed analysis of the requirements contained in the SVG and SMIL Animation specifications. Combing through these specifications I collated the requirements in a database described in Appendix A. My original intention was to be able to link the requirements to test cases and test results in order to ensure each requirement was properly addressed. To my disappointment I discovered that most of the freely available

products for managing requirements were very weak in the area of traceability to test cases.

I set about writing my own database but ultimately this proved too large an undertaking for this project and I could only afford time to produce the requirements database with very basic links to test cases and not the strict traceability I had envisioned. It is surprising to me that such tools are not already available, particularly when there is so much potential for a tool to provide benefits such as automated regression testing and enforcing traceability of requirements to test cases.

Appendix A contains a report from this database.

### 3.9 Design strategy

Based on the requirements database and ideas I had developed through the two prototypes I produced an initial design. After revising this design several times I considered existing implementations.

During this stage I reviewed Schmitz's (n.d.) design proposal for incorporating SMIL Animation into another SVG implementation: Apache Batik. This design contained many useful ideas and so I synthesised my current design with Schmitz's. The design as I have implemented it is closer to Schmitz's proposal than my original plans. One reason for this is because Schmitz's experience in implementing SMIL Animation in other environments has allowed him to produce a more detailed and mature design than I was able to with my limited experience.

Another reason my design closely resembles Schmitz's is because I deliberately chose to follow Schmitz's design closely so that others who might be familiar with Schmitz's proposal could better understand my work. Nevertheless I have deviated from Schmitz's design in several areas which I will describe in Chapter 5.

Typically agile methodologies shy away from detailed up-front design. This is certainly true for Extreme Programming where Beck promotes "designing through

refactoring” (Beck 2000, p. 107). In retrospect I am not certain that this approach would be appropriate for the particular case of SMIL Animation. My experience was that SMIL Animation describes many peculiar edge cases and features that, although appearing quite unassuming on the pages of the specification translate into major design issues. One example is hyperlinking. Although hyperlinking is only a minor feature and receives little attention in the specification implementing this feature invalidates assumptions that might otherwise be made about the direction of time in the model.

While it would be possible to cope with many of these cases by refactoring as they arose, it would result in excessive rework. In this case it is better to have a thorough understanding of the specification and develop a suitable architecture to handle the full complexity of the specification and then refine this through refactoring.

## 3.10 Test strategy

In order to test correct functioning of the implementation, testing was performed on several levels.

### 3.10.1 Unit tests

Early in the project I had expected to do the bulk of my testing through carefully design SVG test cases. These test cases would be supplemented by unit tests to cover some specific cases such as verifying correct parsing. However, during implementation I noticed compile times had increased, perhaps because of changes elsewhere in the codebase. This forced me to rely more heavily on unit testing as it was too inefficient to re-link the entire layout module each time I made a change. With unit testing however I needed only re-compile the files I had changed and link against them.

This change in approach worked to my advantage as I quickly built up a suite of automated test cases that I could run every time I made a change giving me more



confidence that each time I made a change I was not adding a regression. Both Beck (2000, p. 115) and Fowler (1999, p. 89) describe the value of automated unit tests. Beck in particular extols the benefits of this approach for boosting the programmer's confidence. This approach also greatly improved the quality of the code I produced as I frequently produced unit test cases that revealed real defects in the code. A selection of these test cases are included in Appendix B.

In order to isolate components for effective testing I found the approach of using mock objects to be very helpful. Mock objects are a formalisation of an approach that is used by programmers for testing interdependent components by substituting a dependent component with another component that implements the same interface (Mackinnon, Freeman & Craig 2000). This substitute component (the "mock object") can then verify it is called as expected and can produce input to force the test component to traverse certain paths in the code such as error handling code which might otherwise go un-tested.

Frameworks for simplifying mock object development are now starting to appear alongside unit test frameworks although I chose not to use frameworks for either case because this would mean any such tests would be difficult to share with other developers and would certainly be unsuitable for submission to the Mozilla project.

I consider it would be very beneficial to be able to include these unit tests along with my submission to the Mozilla project so that other contributors can ensure that their changes do not introduce regressions and as another source of documentation. However, it seems that this is not common practice in the Mozilla project where unit tests are only included for core functionality such as XPCOM. It may be unreasonable to expect other contributors to also maintain the unit tests when they make changes but I do not think so.

### 3.10.2 SVG test cases

In addition to these unit tests I produced several SVG test cases. These cases are necessary because it is difficult to test the bindings from the SMIL model to

the SVG elements using unit testing because of the many dependencies of SVG on other parts of the codebase. The focus of the SVG tests cases is therefore on these bindings and also on providing further confirmation that the animation is behaving as expected.

The SVG tests cases have been arranged in a manner so that they can be run by alternative SMIL implementations such as the Opera Web browser or Adobe's SVG Viewer. This allowed discrepancies between implementations to be identified which in some cases caused me to reconsider my understanding of the specifications.

Also, by publishing these test cases I was able to receive feedback on their accuracy and in one case I revised the test cases after one reviewer challenged my understanding of one area of the SMIL Animation specification.

### 3.10.3 Public test suites

The SVG working group and the World Wide Web Consortium have been criticised for providing only a small test suite for such a large specification as SVG. Fortunately a much larger test suite is available for SMIL. However, much of the SMIL test suite cannot be readily applied to a project such as this. The reasons are largely technical. For example, most of the test cases do not use SVG as the host language and the test suite is not strict in its use of namespaces. These limitations could be overcome by converting the test suite to a more suitable form but this is an arduous task and has not yet been done. It is also not clear if the result of such work could be re-published for others to use.

For the purposes of this project I converted some of the simpler test cases to a form that was useful for testing and used these to verify my work.

In addition to the test cases I developed for this project several others have developed their own test suites. Hoffmann (2005) presents a particularly thorough test suite along with test results from many current SVG viewers. It is my view that these efforts are of great value to the SVG community, more so than any

one implementation of the specification. Interoperability between implementations is paramount if the Web is to succeed as an application platform. If Web authors spend the bulk of their time attempting to navigate a course between areas of broken behaviour in differing implementations, the Web platform begins to lose the appeal it might otherwise hold as a rapid application development environment. Furthermore, it is often the more advanced and attractive features of the specification that suffer because they are not part of the subset of features fully supported in all implementations. These features are therefore effectively useless and this trend holds back the growth of the Web platform into a more sophisticated framework for application development.

These issues confront all open standards for the Web, however for some other standards specific test suites have gained prominence and taken on the status of un-official measures of the quality of an implementation. For example, the Web Standards Project (WaSP) have created the Acid2 test to test a Web browser's implementation of various Web standards such as HTML4 and CSS1 (WaSP n.d.) and is often used as a benchmark of the compliance of these parts of a browser's layout engine. SVG would benefit from the development of further test suites or from collecting the various test cases from around the Web in a central location.

### 3.11 Communication

I have already mentioned the value placed on transparency in the open-source community. Open-source projects are nearly always collaborative efforts and the commonly held view is that such projects are most effective when information flows freely.

For this reason when I reached the point where I had work to contribute to the project I began by publishing my designs on the Mozilla Wiki Web page. Later I created a personal Web site to report on my progress from which I received important comments regarding my code and tests cases and confirmation of the popular demand for declarative animation to be implemented in Mozilla.

In addition to this, contact with other Mozilla SVG developers via Internet Relay Chat (IRC), Usenet and email provided valuable feedback and information.

By maintaining good communication such a project is more likely to produce a contribution that is accepted by reviewers. Furthermore it also reduces the risk of redundant work being performed by developers who are working in parallel but are ignorant of each others' efforts. As one developer explained to me, "he who implements wins, unless they get into a fighting match with the reviewers".

### 3.11.1 Documentation

A feature of most agile methodologies is that documentation is lightweight compared to the tomes produced under traditional software development models. Instead, communication takes other forms such as test code and face-to-face discussion during pair programming. It is difficult to characterise documentation within open-source projects except to say that it is generally only an adjunct and an afterthought. Some projects do maintain helpful and up-to-date documentation but many more follow the rule "the source code is the documentation".

As many of the documentation techniques of Extreme Programming rely on physical proximity and a small tight-knit team these were not appropriate to my project. However, at the same time I am also not willing to accept that source code is the sum total of a project's documentation. Therefore in addition to the source code I have produced I have documented the code in the manner described above, namely the design descriptions I have posted to the Mozilla Wiki and my personal Web site. This documentation is still lightweight compared to formal specifications characteristic of many software development models and focuses only on those aspects that are expected to be relevant to other developers. Over time I intend to remove details from this documentation in order to extend its lifespan. Once it has served its purpose and is no longer useful it will be removed altogether.

As for the source code itself, I have paid particular attention to ensuring that the code communicates its purpose clearly. I have also based many of the algorithms

on the pseudocode provided in the SMIL Animation specification so that the correspondence between the two is obvious.

### 3.11.2 Risk management

I have already referred to risk management many times in describing the Spiral Development process I have adopted, and risk assessments have been a feature of the status reports I have delivered throughout the project. I will very briefly describe some of these risks as examples of the sort of management required for this project.

The most critical risk to this project was that I might be unable to implement the selected feature to a level that is demonstrable. My mitigation strategy was the iterative development process outlined at the start of this chapter. The success of this approach should be evident from the fact that at the completion of this project the work I have performed is demonstrable and has been so for approximately six months.

Another risk I identified was that another contributor may independently implement the same feature as I have chosen. While this would not make my work entirely redundant, it may lessen its value to the developer community and would seem like an inefficient use of effort. My response to this risk was to communicate my intentions clearly to those who might be considering tackling this topic themselves. This approach was successful and the status log I maintain is referred to by many Web sites that track the progress of SVG in Mozilla.

A final example of the risk management employed in this project concerns the risk that the performance of the final product might be insufficient. I addressed this risk in two manners. Firstly, the iterative strategy I have outlined served to provide early feedback which assured me that performance would be at least tolerable. Secondly, time was allocated later in the project specifically for performance tuning. I followed this plan and was able to successfully improve the performance of the code as described in Chapter 6.

The final evidence of successful risk management in this project is that none of the identified risks eventuated and instead I was able to successfully deliver a working product within the scheduled time.

# Chapter 4

## Analysis

The model defined by the SMIL Animation specification is not trivial and in this chapter the high level requirements and characteristics of this model are described. The other major source of complexity in this project is that this model must be integrated into the Mozilla codebase which, as we shall see, is a challenging task.

### 4.1 Analysis of the SMIL model

I have already described how the features of SMIL Animation can be broken into two models: the timing model and the animation model. This is not obvious from the specification but is elaborated by Schmitz (n.d.). Apart from this broad division of functionality, there are several other distinguishing characteristics of the SMIL model.

#### 4.1.1 Data-type agnostic animation

I have so far referred to the SMIL model without reference to SVG. This is because the semantics defined in the SMIL specification are independent of any particular data types and can be applied to any number of host languages. Examples

of host languages that utilise SMIL are HTML or XHTML—HTML+TIME or XHTML+SMIL—Multimedia Messaging System (MMS) and SVG. As such it is possible to implement a SMIL timing and animation model largely independently of SVG. In the context of Mozilla such an approach is preferable as it may then be re-used in a variety of other host languages supported by the Mozilla platform.

According to the SMIL Animation model, animation elements identify target attributes within target elements and specify how those attributes should change over time. One particular detail that deserves attention is that these attributes may be one of those defined for the target element in the host language's Document Type Definition (DTD) or may be a Cascading Style Sheet (CSS) property of the element. In the case of SVG this distinction is somewhat blurred as in many cases the same attribute fulfils both roles.

Figure 4.1 shows a series of animation elements targeting different attributes on the same target element. The first two elements use the same syntax although the first target attribute is an XML attribute whilst the second target attribute is a CSS property. For the third animation element a different syntax is used. This is one of a handful of special cases in SMIL animation where different handling is provided for some data types. However, even in this special case the animation parameters follow the same syntax and semantics as the preceding elements.

```
<?xml version="1.0" standalone="no" ?>
<svg xmlns="http://www.w3.org/2000/svg" width="200px" height="100px">
  <circle fill="green" stroke="black" cx="50" cy="50" r="40">
    <animate attributeName="cx" from="50" to="150" dur="2s"/>
    <animate attributeName="stroke-width" from="1" to="4" dur="2s"/>
    <animateColor attributeName="fill" from="green" to="blue" dur="2s"/>
  </circle>
</svg>
```

**Figure 4.1:** A set of animation elements targeting different attributes on the same target element.

Generally speaking, the SMIL model is independent of the host language it is integrated with and the data types being animated.



### 4.1.2 Additive composition

SMIL allows for an animated attribute to be the target of more than one animation. For example in Figure 4.2 two animations each target the `cx` ( $x$  centre) attribute. The result is a combination of the two animation functions.

```
<?xml version="1.0" standalone="no" ?>
<svg xmlns="http://www.w3.org/2000/svg" width="200px" height="100px">
  <circle fill="red" stroke="black" cx="50" cy="50" r="40">
    <animate attributeName="cx" by="-50" dur="2s"/>
    <animate attributeName="cx" to="150" dur="2s"/>
  </circle>
</svg>
```

**Figure 4.2:** An example of two animations targeting the same attribute.

SMIL Animation defines rules for when and how these animations are to be combined and how to determine which animation takes precedence if the animations are not to be combined. This suggests some sort of arbitration is necessary to manage the competing animations.

One interesting special case is a particular type of animation known as ‘to animation’ which is described by the specification as being “a kind of mix of additive and non-additive”. The behaviour is quite different from the normal cases and requires particular attention during implementation.

### 4.1.3 Timer independence

Times in SMIL are defined independently of any frame rate. Given the wide range of hardware on which Mozilla is expected to run, this implies that no assumptions can be made about the amount of time that has elapsed between successive samples. On one platform this may be a matter of milliseconds and the the new sample may be only slightly different to the last. On another platform whole seconds may elapse between frames and in this time several animation

intervals may have begun and finished and changes may have been made to the animation parameters using script. All this requires that the timing logic be completely independent of the frame rate.

A further consideration is that document time is not continuous. It is possible through features such as hyperlinking to ‘seek’ the animation to a point in the past. This requires further specialised handling if assumptions are made about the direction in which time progresses.

#### 4.1.4 Dynamic document model

One of the most challenging aspects of implementing SMIL Animation in SVG is that SVG fails to disallow changing of the animation parameters at runtime. Although the SMIL Animation specifications states, “Language designers integrating SMIL Animation are encouraged to disallow manipulation of attributes of the animation elements, after the document has begun.” the SVG 1.1 specification does not provide any such restriction. In fact it even refers to using DOM methods to modify animation elements.

Allowing such changes is a significant consideration for any implementation. In fact, the SMIL Animation specification provides, “Dynamically changing the attribute values of animation elements introduces semantic complications to the model that are not yet sufficiently resolved”.

In general, this will mean that fewer optimisations are possible as state necessary for resolving changes to the model must be maintained. Furthermore, the logic for performing a minimal update to the model will differ depending on the nature of the change and is therefore likely to be complex.

#### 4.1.5 Dynamic dependencies

Another feature of the SMIL model that will have a considerable impact on any design is that animations may be defined to be relative to one another. This

implies that animations must communicate changes in state to one another and the specification defines this behaviour for a range of different situations. This is a significant source of complexity in the model.

## 4.2 Analysis of the Mozilla codebase

Writing a completely new software system is an entirely different matter to integrating some new functionality into an existing system, especially when the system to be integrated with is as complex and established as the Mozilla codebase.

Before undertaking this project several people, including some from within the SVG developer community warned me about the difficulties of comprehending the Mozilla codebase for a short-term project. This was a fair warning and even after eighteen months of working with this codebase I still could not present a comprehensive overview of the platform. Fortunately this is not necessary as this task has been undertaken successfully by many others who are far more capable than I of explaining the many components and technologies that make up Mozilla (in particular see McFarlane 2003*b*, Oeschger, Murphy, King, Collins & Boswell 2002, Dsouza, Hildebrand & Israeli 2004). Instead I will describe only those components that are related to this project and even here I will restrict my description to very general terms.

### 4.2.1 Netscape Portable Runtime

The Netscape Portable Runtime (NSPR) is the basic platform abstraction layer upon which all other components are built. This layer provides basic services such as threading and synchronisation, timers, fixed-size data types, data structures such as hash tables, file and network input and output, and memory management.

NSPR itself has only a small impact on a project such as this. The only effect it has is that programming code should be written to make use of the services

provided by NSPR to ensure consistent behaviour across all the platforms that Mozilla supports. However, the presence of NSPR is a reminder of the cross-platform nature of Mozilla and this does have a significant impact on this project.

NSPR currently supports Windows platforms, Macintosh and at least twenty versions of UNIX (mozilla.org 2000) and the Mozilla codebase as a whole also supports these platforms to varying degrees. In order to ensure that code written for Mozilla can be used across as many platforms and compilers as possible a detailed set of portability guidelines has been established. These guidelines preclude many C++ features such as exceptions, run-time type information, C++ standard library features and many uses of templates (Williams 1998). Generally, each of these are disallowed because of inconsistent compiler support for the feature. One impact of this emphasis on portability is that it limits the range of third party libraries that can be used. In some cases this has been addressed by writing new utilities that comply with the requirements of the Mozilla platform.

This requirement for cross-platform portability will have a significant effect on the design of any new features added to the Mozilla codebase and must certainly be considered in this project.

### 4.2.2 XPCOM

In order to provide cross-platform component technology the Mozilla project has developed the Cross-Platform Component Object Module (XPCOM). This technology is similar to other component technologies such as Microsoft COM and, to a lesser extent, CORBA. Such technologies promote greater modularity and decreased coupling between software components by providing facilities for components to communicate via interfaces without being aware of the implementation of those interfaces. This includes being aware of the programming language used to implement the interface.

For the case of XPCOM, most components are implemented in C++. In this context XPCOM represents a different approach to object design. Because C++

does not have the concept of interfaces such as are used in Java<sup>1</sup> complex class hierarchies often involve multiple inheritance with virtual base classes and run-time type information (RTTI). Compared to this, XPCOM avoids virtual base classes and uses `QueryInterface` in place of RTTI and `dynamic_cast`ing between interfaces. This has significant effects on the arrangement of class hierarchies.

XPCOM has its costs and is not without its critics (Collins 2002). One reason is the performance costs associated with the sometimes unnecessary layers of abstraction it introduces. For example, all method calls through an XPCOM interface are virtual and the cost associated with this can, when performed frequently enough, be significant. For this reason there is an ongoing effort to remove XPCOM usage where it is not useful, a process known as de-COMtamination. Avoiding unnecessary use of XPCOM is therefore another design consideration.

### 4.2.3 Utility of SMIL

The Mozilla platform is capable of processing many languages other than SVG. Given that SMIL Animation is defined independently of any particular host language it would certainly be beneficial to the Mozilla platform to implement SMIL independently of SVG. Taking this approach would allow SMIL support to be later extended to XHTML to support the XHTML+SMIL Profile for example.

### 4.2.4 The creation of a platform

In the background to this project I have described how the Mozilla codebase provides an attractive platform for application development. This emphasis on developing an application platform implies a certain approach to implementation. In a typical SVG implementation produced simply to render graphics, qualities such as functional correctness, performance, and rendering quality are important.

---

<sup>1</sup>Although pure abstract classes are possible, C++ does not distinguish between this case and regular abstract base classes.

However, in developing an application platform, scriptability—the ability to update the model dynamically—is also of great importance. Similarly, integration with the remainder of the platform is also a high priority.

### 4.2.5 Contributing to a mature product

The Mozilla codebase is used to develop high-profile mass-market products such as the Firefox Web browser. For this reason additions to the codebase are strictly controlled by a review process. This reviewing seeks to ensure that the code being added does not introduce new security flaws, is of high quality, is consistent with the remainder of the codebase and does not increase the code size of the final product unnecessarily.

Addressing these concerns in the design of any new contribution may increase the likelihood and rate at which it is accepted. For example, by structuring the code such that it can be conditionally disabled, it may be able to be accepted into the Mozilla codebase in a disabled state from which point outstanding concerns can be addressed.

# Chapter 5

## Design

Based on the analysis of the requirements for this project we now consider an overall approach to meeting these requirements, the broad functional components of this design and the process used to arrive at these.

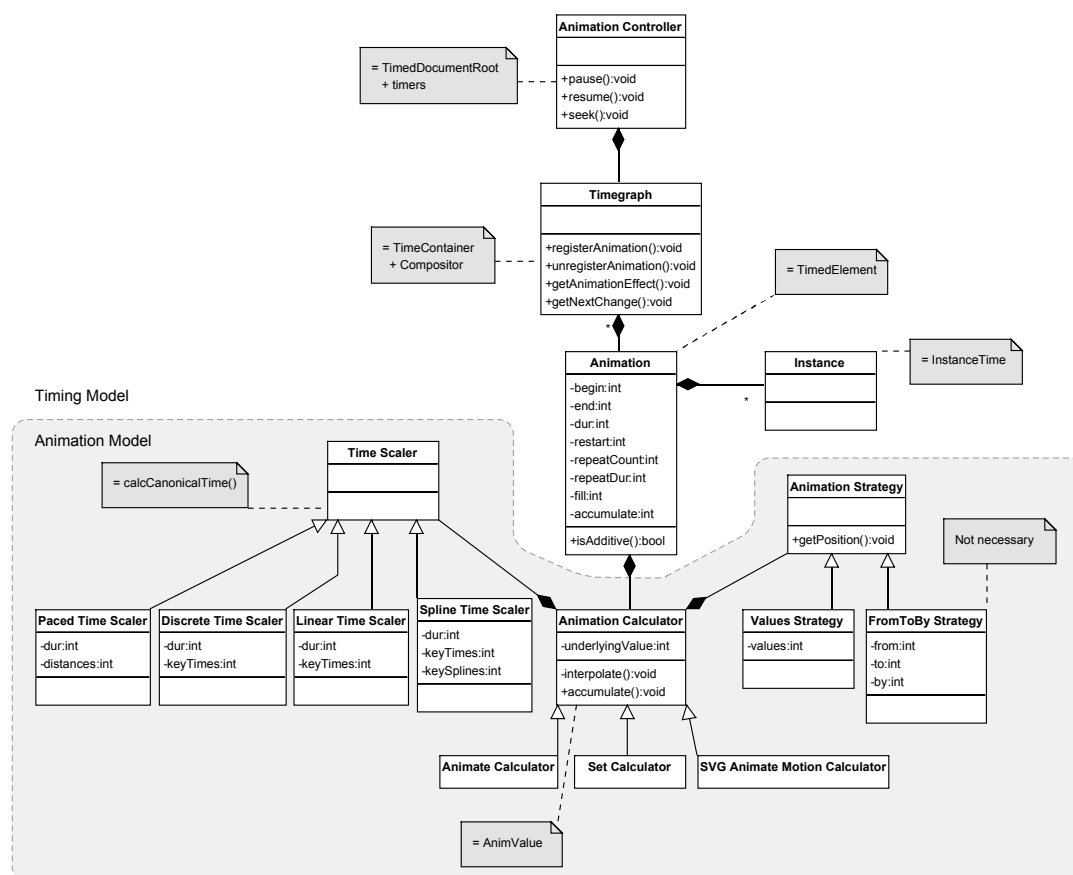
### 5.1 Design process

My initial design efforts followed the lines of typical Object-Oriented Design (OOD). The process I adopted as recorded in my engineering log was:

1. Define the top level objects.
2. Assign responsibilities for the requirements to each.
3. Determine the informational dependencies between each object. In particular, consider how easy it will be to support updating via the DOM. For each property that can be changed, consider how the change would be propagated.
4. Determine ownership relationships.

5. Evaluate the design considering the following questions: Is it testable? Will it be possible to extract useful subsets for each iteration without implementing the whole thing?
6. Iterate.

Using this approach I developed the design shown in Figure 5.1.



**Figure 5.1:** Class diagram of the design prior to incorporating Schmitz's ideas. The annotations and division between timing and animation model were not present in the original design but have been included here to show the correlation between this design and the design shown in Figure 5.2.

Following on from this design I investigated existing implementations and related patterns. During this stage I discovered Schmitz's (n.d.) detailed design document for supporting SMIL Animation in the Apache Batik SVG toolkit. Schmitz



is one of the editors of the SMIL Animation specification and his design is based on experience in implementing this standard.

Schmitz introduces the division of the model into those components relating to timing and those relating to animation. Between these two only simple times and minimal state information is exchanged. In addition to establishing this separation of concerns Schmitz describes the components within each of these two parts. Details of integration with Batik such as timers for producing frames are deliberately brief to emphasise the independence of the model from any particular application or sampling frequency.

There are many advantages to reusing Schmitz's design. Firstly, it serves as an additional source of documentation of my work. Secondly, it provided me with greater confidence in the design allowing me to spend less time on design in successive iterations. Finally, this kind of reuse saves time and is good software engineering practice. Schmidt & Stephenson (1994) take up this topic promoting design patterns as a means of achieving reuse across operating system platforms. Although Schmitz's design, even without the specific references to Batik, may be too application-specific to fit Gamma, Helm, Johnson & Vlissides's (1995, p. 3) definition of a software design pattern, design reuse is still a valuable practice. In his assessment of Gamma et al.'s (1995) *Design Patterns*, Vinoski asserts that reusable design is 'the real key to software reuse'.

After reviewing Schmitz's proposal I highlighted ideas to incorporate into my own design. However, by the end of this process my design resembled far more closely Schmitz's proposal than my original starting point. This was partly due to a deliberate attempt to base the names of classes and properties on those in Schmitz's design to highlight correspondence between my design and Schmitz's proposal. There are however several notable deviations from Schmitz's proposal which I will outline in the corresponding description of each component.

The result of the revised design is shown in Figure 5.2.

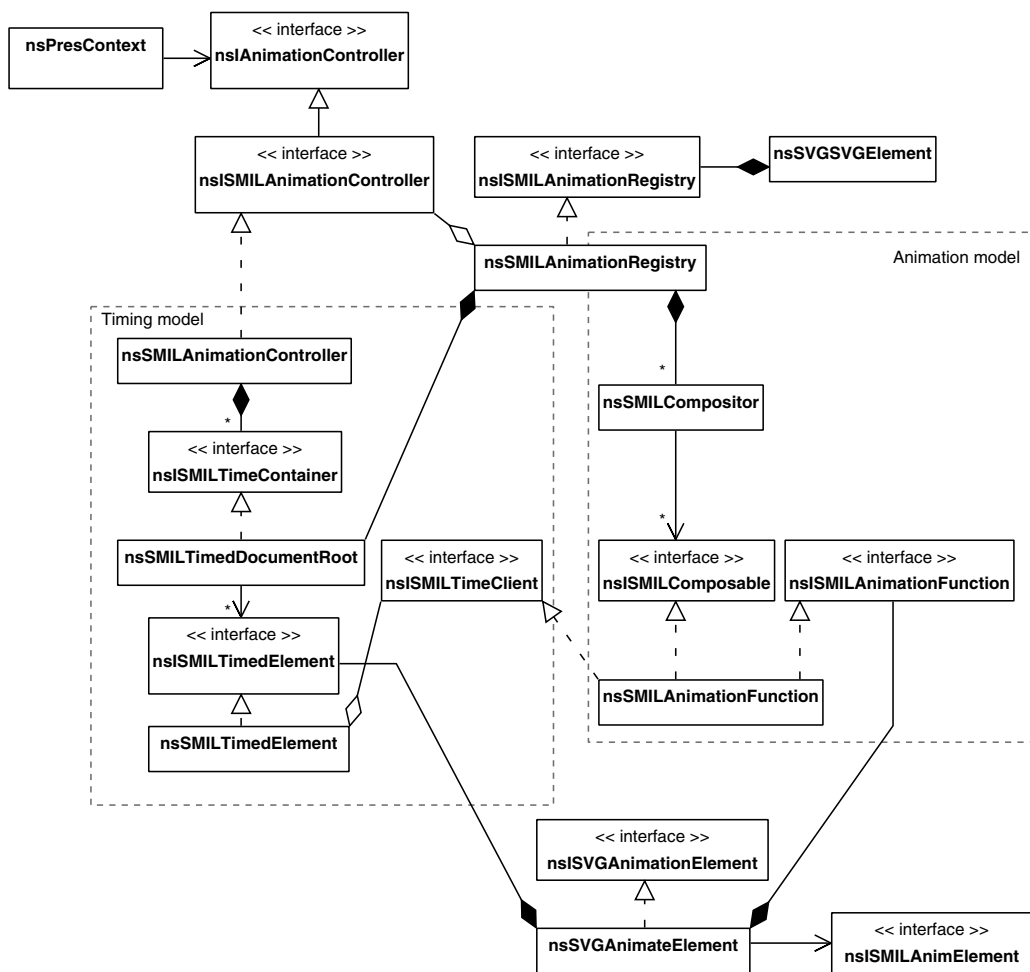


Figure 5.2: Overview class diagram after incorporating Schmitz's ideas.

## 5.2 Operation overview

The overall flow of control in this design is based upon three stages:

- document setup,
- sampling, and
- composition.

### 5.2.1 Document setup

When an SVG document fragment is loaded, an `nsSVGSVGElement` is created by the parser. This object creates the animation controller if it does not already exist and associates it with the current presentation context.

When an animation element is loaded, an appropriate SVG object is created. For an `<animate>` element this is `nsSVGAnimateElement`. This object requests the animation registry object which is created if necessary. There is one registry per SVG document fragment. This registry is then used to register with the timing and animation model.

The `nsSVGAnimateElement` is represented in the animation model by an `nsSMILAnimationFunction` and in the timing model by an `nsSMILTimedElement`. All the parsing work including supplying default values is performed by these two objects. This simplifies `nsSVGAnimateElement` considerably and allows this functionality to be re-used by other animation elements such as `<animateColor>` and even in other host languages for SMIL.

### 5.2.2 Sampling

The animation controller is responsible for the animation timer and periodically instructs its time containers to conduct a new sample. The time containers in

turn call their timed elements. The timed elements compare the sample time to their parameters and if they are active supply their time client with a new sample time adjusted according to their parameters. The animation functions which act as time clients store the sample parameters provided by the timed elements.

### 5.2.3 Composition

At the end of a sample, the time containers instruct the registry with which they are associated that the sample has ended and at this point composition is initiated. The `nsSVGSVGElement` is informed via the `nsISMILAnimationObserver` interface (not shown) that composition is beginning so that redrawing may be suspended.

There is one compositor object for each animated attribute and each is called in turn. Each compositor sorts its animation functions according to their priority and then calls each function to build up the final animated value which is then set on the appropriate SVG element.

At the end of sampling the `nsSVGSVGElement` is informed that composition has finished at which point redrawing is unsuspending and the display is updated.

A summary of the sampling and compositing steps is shown in the sequence diagram in Figure 5.3.

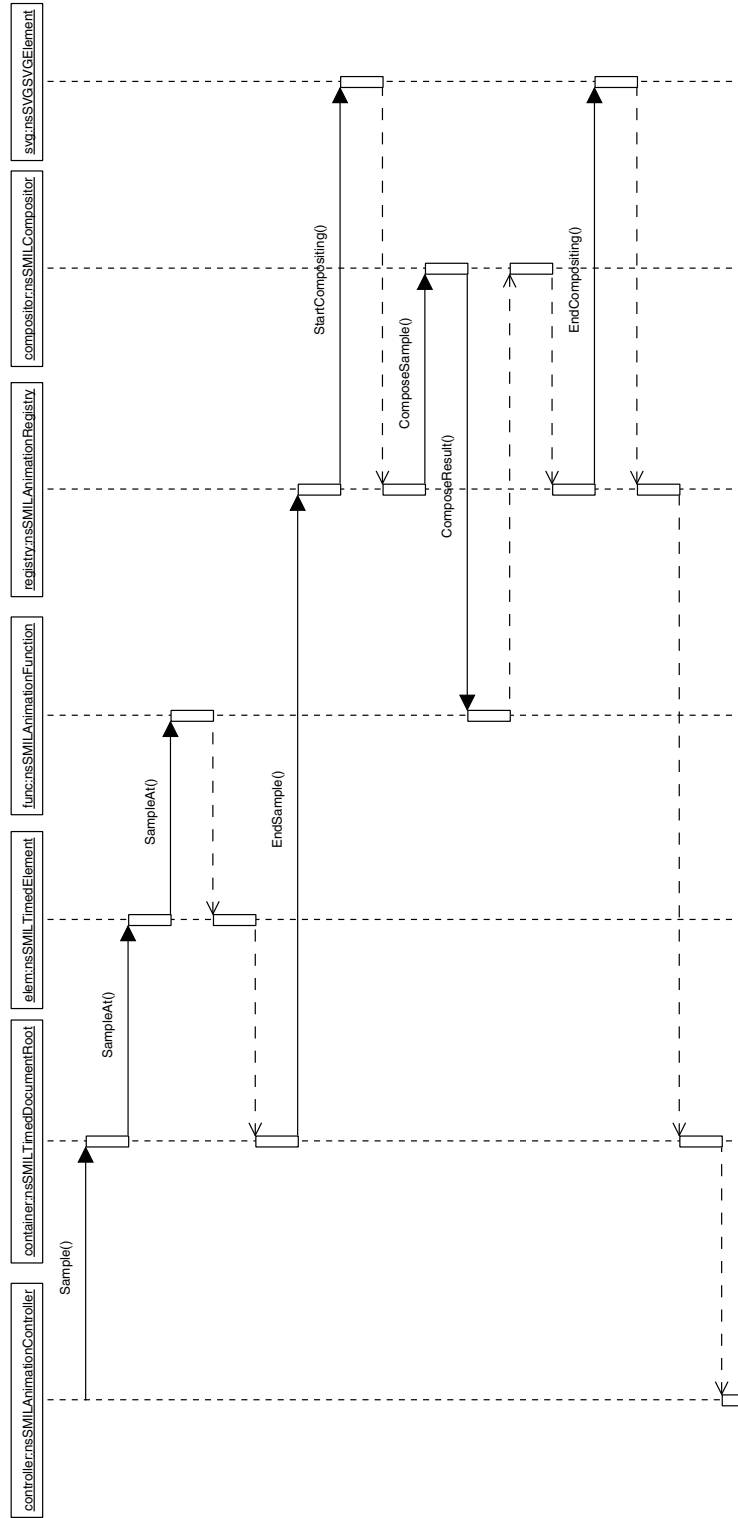


Figure 5.3: Sequence diagram for a single animation sample.

## 5.3 Comparison with Schmitz's model

From this high-level view the most significant deviation from Schmitz's model is the distinction between sampling and composition and the inclusion of the animation registry. The reasons for this discrepancy are detailed in the description of the animation registry.

Another difference relates to the use of object composition rather than inheritance to endow the object representing the `<animate>` element with timing and animation support although Schmitz does refer to an implementation in terms of delegation. Object composition is sometimes favoured as providing better encapsulation, reducing coupling and providing better runtime flexibility (Gamma et al. 1995, pp. 18–20). In this case the reduced coupling and increased encapsulation assist in producing a more self-contained SMIL implementation that can be easily disabled.

Many of the other differences between this design and that proposed by Schmitz are not visible from this high-level view and will be covered in the description of each component.

## 5.4 Timing model

A class diagram of the components that make up the timing model is shown in Figure 5.4. I will describe the details of each class in this diagram in turn.

### 5.4.1 Animation controller (`nsSMILAnimationController`)

The animation controller (shown only in Figure 5.2) does not belong to either the timing or animation model but rather drives them both. The animation controller maintains the animation timer so it is this class that determines the sample times and sample rate.

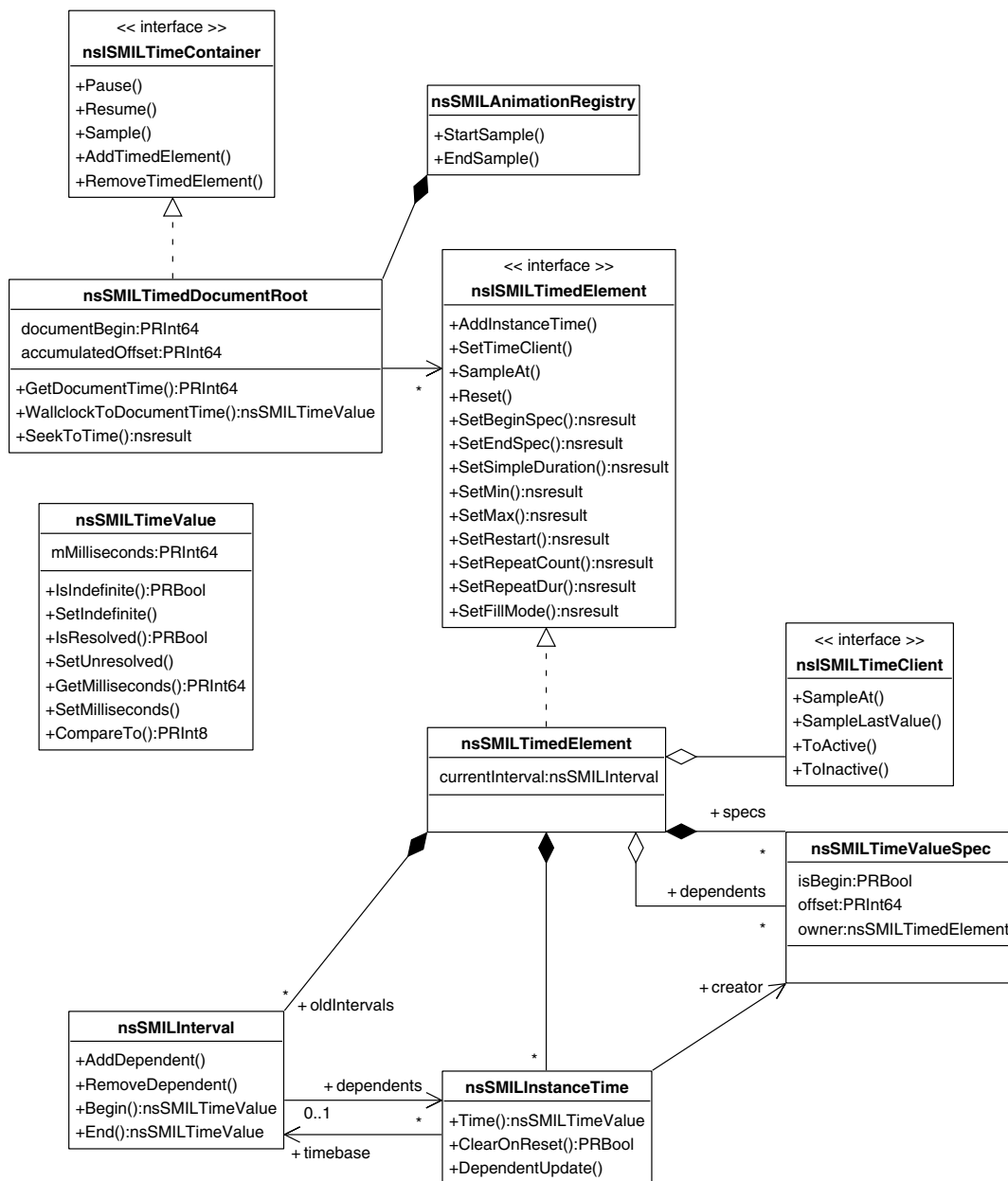


Figure 5.4: Class diagram of the timing model.

There is at most one animation controller associated with a presentation context. The implication of this is that for an XHTML document containing several independent SVG fragments, they will all be driven by the same controller. This arrangement allows frame-rate tuning to be performed at a document-level which is likely to produce better results than attempting to tune several controllers independently.

When a document containing SVG is loaded the `nsSVGSVGElement` checks if there is already an animation controller associated with the presentation context and if there is not, a new controller is created and associated. The controller object is required even for un-animated SVG in order to initialise the timed document root with the appropriate start time for the document in case animation is later added via script.

#### 5.4.2 Timed document root (`nsSMILTimedDocumentRoot`)

The timed document root is a concrete implementation of the time container interface that translates samples into times relative to the start time of the document. There is one such object per SVG document fragment. This class is responsible for notifying the registry when a new sample begins and ends.

The time container interface hides the details of the timed document root from the animation controller and is a step toward implementing the kind of time container support described in the SMIL 2.0 specification of which SMIL Animation is a subset.

#### 5.4.3 Timed element (`nsSMILTimedElement`)

A timed element contains the timing properties of an animation element such as `<animate>`. This is perhaps the heart of the timing model as it is here that most of the rules for calculating new intervals, implementing fill modes, repeating and providing restart behaviour are implemented.



#### 5.4.4 Time client interface (`nsISMILTimeClient`)

The time client interface is the primary interface between the timing model and the animation model. Currently only one time client is supported per timed element. This may be extended in order to support `<animateMotion>` which may, for example, be modelled as one `nsSMILTimedElement` with three time clients corresponding to the `x`, `y`, and `transform` attributes of the target element. Most likely though this would be implemented by animating a single transformation matrix.

#### 5.4.5 Time value specification (`nsSMILTimeValueSpec`)

A time value specification performs parsing of individual begin and end times. SMIL Animation defines several kinds of these specifications such as offset times, wallclock times, event-base times and syncbase times. Currently only offset times have been implemented. Upon implementing other types it may be more suitable to refactor this class into an interface with several concrete implementations and a static factory method in the interface to create the appropriate object. In keeping with Beck (2000, pp. 103–107) and Fowler’s (1999, p. 58) guidelines I have deferred this refactoring until it is necessary.

#### 5.4.6 Interval (`nsSMILInterval`)

An interval is essentially a structure consisting of a begin and end time. It is used for representing the current interval and also for storing past intervals for the purpose of hyperlinking back in time. For syncbase timing, intervals also inform their dependent instance times when they have been updated.

#### 5.4.7 Instance time (`nsSMILInstanceTime`)

An instance time represents an instant in document simple time that may be used to construct a new interval. Instance times created by time value specifications are also responsible for informing their creator when they have changed.

This might happen when the instance time corresponds to a synchbase time value specification and the interval on which it depends changes.

#### 5.4.8 Time value (`nsSMILTimeValue`)

A time value is a simple tri-state data type. Times are expressed as a signed 64-bit integer representing milliseconds or the special values *indefinite* or *unresolved*. The use of 64-bit integers rather than floating pointer numbers simplifies some comparisons and should provide improved performance, particularly as 64-bit processors become more prevalent.

A 32-bit signed integer was considered to have insufficient range, allowing a range of only 25 days in either direction. While this may seem to be ample for an application that typically only animates for a matter of seconds before being replaced with other content, we need to consider the full range of use cases for the Mozilla platform. Amongst these are various embedded applications such as the Minimo project that seeks to produce a lightweight version of Mozilla suitable for small devices. Consider a scenario where Mozilla's layout engine—Gecko—is embedded in an informational display at a railway station showing the arrival and departure times on each platform. In this case it would be unacceptable for the application to fail every 25 days because of an inadequate representation of times. The use of signed 64-bit integers in this design provides a range of 292 million years in either direction which should be sufficient for most applications.

### 5.5 Animation model

A more detailed view of the classes that make up the animation model component of the design is shown in Figure 5.5. We will discuss each class in this diagram in turn.

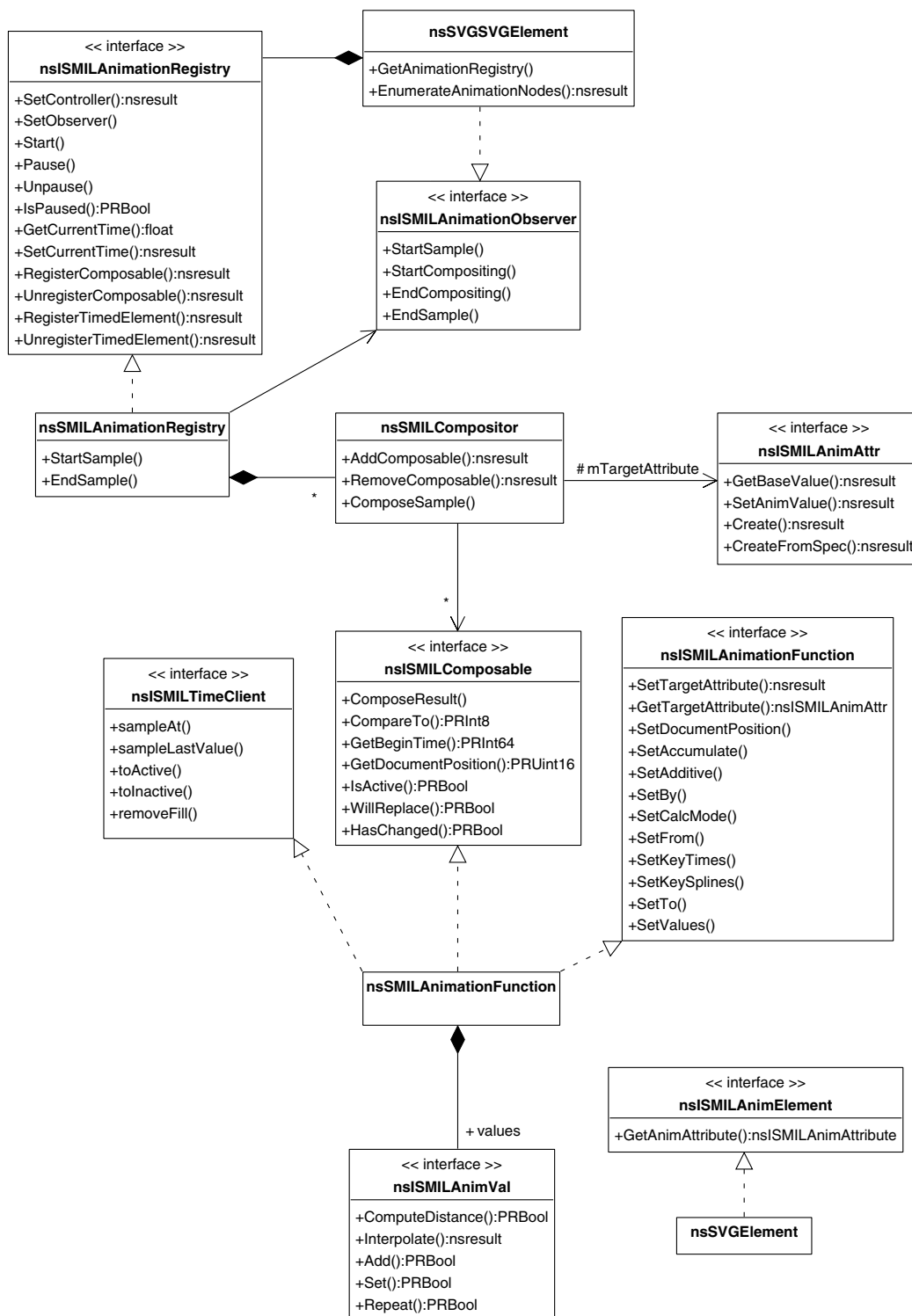


Figure 5.5: Class diagram of the animation model.

### 5.5.1 Animation registry (`nsSMILAnimationRegistry`)

The animation registry acts as a single point of registry with the SMIL model. There is one such registry per SVG document fragment. This class is not present in Schmitz's design. I have included it for three reasons.

Firstly, the animation registry simplifies registering for animation elements and the outermost SVG element. Due to the way compositing is performed it is necessary for each animation element to make itself known to both the time container for the SVG document fragment and the compositor assigned to the attribute being animated. To simplify these operations and hide some of the details of the SMIL model, the animation registry provides a simple interface for registering with the timing and animation model.

Secondly, the animation registry allows per-sample operations to be performed at the appropriate time. Schmitz's model does not delve into integration issues such as suspending and un-suspending redrawing so I have chosen to include this here via the `nsISMILAnimationObserver` interface. This interface provides a few methods called at pertinent times so that operations such as suspending and un-suspending redrawing can be performed.

Finally, the animation registry allows the compositing to be controlled 'from above'. This is the most significant deviation from Schmitz's design. In Schmitz's design the timing and animation model are very elegantly kept at arm's length through the time client interface. As each composable object is sampled it provides its result to the compositor. The compositor tracks the results it has received and when it has received results from all the registered active composable objects it begins composition. Some exceptions have to be accounted for such as 'to animation' and relative values and for this Schmitz suggests callbacks could be used.

The design presented here operates in the opposite direction. The composable objects simply store the sample parameters provided through the time client interface. These parameters include information such as the simple time of the last sample. After all composable objects have been sampled the *animation*

*registry* is told to start compositing. The animation registry instructs all its compositors to perform compositing. Each compositor then iterates through the composable objects requesting their results as necessary.

Some of the advantages of this approach are:

- No special handling is required for ‘to animations’.
- The compositor does not need to combine results, or even know about the additive behaviour of its composable children.
- The compositor is free to optimise as it sees fit by only requesting those composable objects that will actually affect the final result calculate their results.
- Relative values can be recalculated in a simpler fashion.
- Animations that are frozen do not require sampling as they will simply reuse the parameters passed to them at the final sample.
- Knowledge of how different types of animations prioritise is confined to the composable objects themselves and not the compositor.

The main disadvantage of this approach is coupling between the timing model and animation model. This coupling appears between the animation registry and the timed document root. However, my view is that the simplicity afforded by this approach warrants the extra coupling.

The animation registry also provides the implementation for several animation related methods of the `SVGSVGElement` DOM interface.

### 5.5.2 Animation observer (`nsISMILAnimationObserver`)

This interface allows a client to be informed of steps in the animation process. This is used by `nsSVGSVGElement` to suspend and un-suspend redrawing before and after compositing as well as to batch enumerating the animation nodes.

Without batching the enumeration which is needed for proper prioritisation of animation functions would occur each time an element was attached to the SVG document tree. If several elements were attached simultaneously such an operation would be of order  $O(n!)$ .

The use of the Observer pattern allows the SMIL model to remain independent of SVG or any other host language.

This interface is not in Schmitz's design. I have included it for reasons which are explained in the description of the animation registry above.

### 5.5.3 Compositor (`nsSMILCompositor`)

Each compositor manages a collection of animation functions that target the same attribute. Each of these animation functions implements the `nsISMILComposable` interface. The compositor is responsible for calling these objects in order from lowest priority order to the highest priority according to the animation sandwich model defined in the SMIL Animation specification.

Each time an composable object is called it is passed the underlying value of the sandwich to which it may add its result or replace it depending on the additive behaviour of the animation function.

The compositor is responsible for re-compositing when a relative value changes and performs optimisations such as filtering out those objects that will not contribute to the final result.

### 5.5.4 Animation function (`nsSMILAnimationFunction`)

This class provides the calculation of animation values for animation elements that perform interpolation such as `<animate>` and `<animateColor>`. This includes providing different interpolation modes, additive modes and time scaling operations.

Later when the non-interpolating `<set>` element is implemented, this class and interface may be split into `nsSMILSimpleAnimFunc` and `nsSMILInterpolatingAnimFunc`. Other animation elements such as `<animateTransform>` and `<animateMotion>` may be implemented as subclasses of this class or by adding extra parameters.

All attribute parsing and handling such as providing default values is performed by this class. This allows this logic to be shared between all animation elements.

This class corresponds to `InterpolatingAnimElement` in Schmitz's design.

### 5.5.5 Animated value interface (`nsISMILAnimVal`)

This interface is the basic layer of indirection used by the animation model to manipulate different data types. The methods in this interface allow all the necessary calculations such as addition and repetition to be performed.

### 5.5.6 Animated attribute interface (`nsISMILAnimAttr`)

This interface sits above the animated value interface to wrap the animated and base value of an attribute together for querying by SMIL. Because this interface provides the association to a particular animated attribute, the animated value objects can be completely independent of any SVG data type and can therefore be very lightweight objects. This is important as many animated value objects are created by the animation function.

### 5.5.7 Animated element interface (`nsISMILAnimElement`)

This interface is not used within the SMIL module but provides a consistent manner for identifying elements that have attributes that can be animated and for accessing those attributes. This interface could potentially be used to allow elements from different XML namespaces but within the same document fragment to be targeted for animation.

Any target element may disallow animation of a particular attribute by returning a NULL result for this attribute.

## 5.6 Design evaluation

The design proposed in this chapter exhibits the following qualities:

**Maintainability.** By basing the design and class and method names on the design proposed by Schmitz the design benefits from the additional documentation provided by Schmitz. This will assist those maintaining the code to comprehend its operation.

**Modularity.** The components in this design are independent of any particular host language. All interactions with the SVG model take place through abstract interfaces such as the animation observer interface. This allows the entire SMIL module to be easily disabled which reduces the risk of accepting it into the Mozilla codebase.

**Reusability.** The use of composition to provide animation elements with animation and timing behaviour as well as parsing support and parameter validation allows this functionality to be easily provided to other animation elements including animation elements in host languages other than SVG.

**Simplicity.** The distinction between the timing and animation model allows the most complex parts of SMIL Animation to be implemented in quite independent classes, namely the timed element class and the animation function class. This avoids the complexity that would result from attempting to implement this behaviour in a single Blob entity (Akroyd 1996).

**Performance.** Although performance is not a major feature of this design the inclusion of the animation observer interface allows rendering updates to be batched which dramatically reduces the amount of busy work that would otherwise be involved in rendering partial frames.



Another performance characteristic of this design is that each sample is conducted synchronously. This means that after the animation controller initiates a new sample, it does not receive control again until the next frame in the animation has been rendered. This allows the animation timer to be tuned according to the actual time used to produce each frame. This gives more useful feedback than if rendering were performed asynchronously in which case the timer could only be tuned on the time required to produce the parameters of the next frame which is roughly constant.

A third performance characteristic is the inclusion of methods in the composable interface to allow the compositor to filter out animations that will not affect the display. This is discussed in the next chapter on implementation details.

# Chapter 6

## Implementation details

The implementation I have produced conforms to the design I have outlined in the previous chapter. However, in implementing this design there are several details that deserve special comment.

### 6.1 Controlling the animation start-point

Although starting the animation might seem to be a simple operation it was complicated by two factors.

Firstly, SVG defines that animations occur at the same moment as the `SVGLoad` event is fired. The timing of this event depends of the use of the `externalResourcesRequired` attribute which is not yet implemented in Mozilla. My approach was to listen for this event so that when the `externalResourcesRequired` attribute is implemented animation will automatically produce the correct starting behaviour. This was simple to implement requiring only trivial changes to the circumstances under which the `SVGLoad` event was dispatched.

One drawback of this approach is that if an SVG document fragment is created entirely by script the `SVGLoad` event will not be dispatched and animation

will never start. This situation will need to be re-considered when support for `externalResourcesRequired` is introduced.

The second source of complication in starting animation arose late in the project when a new caching mechanism was introduced for pages in the Web browser's back and forward history. This mechanism keeps certain documents fully assembled in memory so that when the user navigates backward and forward the document can be restored almost instantly.

One implication of this caching strategy was that animation would continue even whilst the document was not visible which would result in busy work and failed assertions as some objects necessary for animation would be no longer available.

Another implication is that when a document is loaded from this cache it is restored in the state in which it was previously seen. Animations however should not follow this behaviour but should restart.<sup>1</sup>

Several solutions to this problem were trialled. The simplest solution involves creating an unload handler. Documents that have scripts attached that perform actions when the document is unloaded are not stored in the cache as it will not be possible to correctly restore the document to its previous state in these circumstances. The simplest solution therefore would be to associate a dummy unload handler with any document that contains animations. The drawback of this approach is that such documents can no longer take advantage of the cache.

Another alternative is to use the `SVGLoad` and `SVGUnload` events to restart animations but in some circumstances these events are not dispatched<sup>1</sup>.

The approach that is currently implemented relies on registering for page show and hide events that are dispatched when the document is restored and hidden. This approach produces the correct behaviour but further testing has since revealed that these events are dispatched only once for SVG documents embedded via the `<object>` element. It is not yet clear if this is the intended behaviour or if it is a defect in the page transition event dispatching.

---

<sup>1</sup>This is because animations are defined to start at the same time as the `SVGLoad` event is fired and this event should be fired even when the document is loaded from the backward/forward cache although in my testing it is not fired in some circumstances.

One advantage of using page hide and show events is that it reduces the coupling between other parts of the Mozilla codebase and the SMIL module. Those parts of the code responsible for saving and restoring documents do not need to be aware of the presence of SMIL but simply need to dispatch the appropriate events to their registered listeners.

## 6.2 Integrating SVG data types

Perhaps the most disappointing part of the project is that despite developing a useful SMIL model, it cannot not be applied broadly due to pending rework of parts of the SVG model. This rework relates to how to store animated values separately without reserving memory for attributes that are never animated. The issue has been under discussion within the Mozilla developer community for nearly twelve months but is yet to be resolved. The latest indications suggest that this may be tied in with other refactoring of the layout engine that could take several months or more to complete.

The implication of this situation for my project is that there is currently no generalised mechanism for storing animated values which prevents any animation from being performed. In order to continue with development I have produced a naïve implementation for one data type—SVG lengths—that simply stores the animated value separately for each attribute. This allows animation to be performed on attributes that have a length data type such as the dimensions of rectangles or circles. Other attributes cannot be animated.

Earlier in the project I developed a prototype of a suggestion proposed by another member of the SVG developer community. This approach exploited the C++ vtable for the animated object to make one object appear to be two separate objects with the same interface whilst still allowing the object itself to provide different behaviour to different callers. This approach addressed the issue of unnecessary storage but it was not without fault. Current discussion seems to be moving away from this approach and I have removed the code from

my implementation replacing it instead with the naïve approach I have already described.

It would be possible to apply this naïve approach more widely in order to support a wider range of animations but such work would be wasted effort once the proposed rework is completed.

### 6.3 Frozen ‘to animation’

During implementation I encountered several difficulties with the SMIL Animation specification. Some of these appear to be contradictions in the specification which I have begun to document on a Web site that I maintain to report the status of this project. In other cases the behaviour is unexpected or difficult to implement correctly.

The reader will most likely not be interested in these specific situations so I will only briefly describe one difficulty as an example of the sort of complexity that was commonly encountered during this stage.

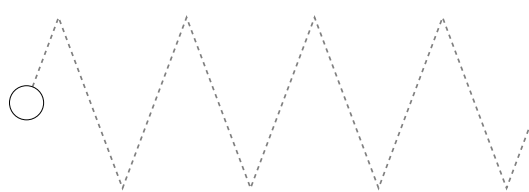
One area in which the SMIL Animation specification is clear but difficult to implement is frozen ‘to animation’. ‘To animation’ is a particular type of animation that features an unusual combination of interpolation and addition. This situation arises when an interpolation interval is defined as having an absolute end value—specified by the `to` attribute—but no starting value. The intention of such an animation is that over the interval the result of the animation should converge upon the specified end value regardless of the underlying value. This can produce interesting effects when combined with other lower priority animations.

Figure 6.1 shows a simple animation where a circle follows the triangle wave indicated by the dotted line. Figure 6.2 shows the result when a higher-priority ‘to animation’ is combined with the animation producing the triangle wave. The resulting motion of the circle as indicated by the dotted line shows how the ‘to animation’ comes to dominate the underlying triangle wave animation.

```

<circle r="10">
  <animate
    attributeName="cx" to="300"
    dur="8s" fill="freeze"/>
  <animate attributeName="cy"
    values="0; 100; 0"
    repeatCount="4.75"
    begin="-1.5s" dur="2s"
    fill="freeze"/>
</circle>

```

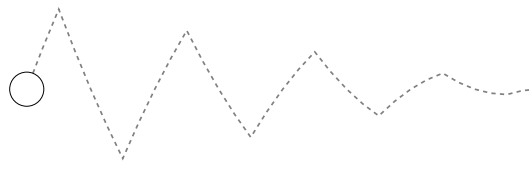


**Figure 6.1:** A simple animation that causes a circle to follow a triangle-wave pattern.

```

<circle r="10">
  <animate
    attributeName="cx" to="300"
    dur="8s" fill="freeze"/>
  <animate
    attributeName="cy"
    values="0; 100; 0"
    repeatCount="4.75"
    begin="-1.5s" dur="2s"
    fill="freeze"/>
  <animate attributeName="cy"
    to="50" dur="8s"
    fill="freeze"/>
</circle>

```



**Figure 6.2:** An animation composed of the triangle wave from Figure 6.1 combined with a ‘to animation’.

This effect is not difficult to produce and in fact the SMIL Animation specification provides a simple formula describing the result. However, this particular type of animation also features a particular type of fill behaviour which I will now describe.

When an animation has completed its animation interval, it may either cease to affect the target attribute in which case this property will appear to reset, or it may hold the target attribute at the final value with which it was animated. This difference is referred to as the fill mode. The first possibility corresponds to the fill mode of **remove** whilst the second possibility where the target attribute is held at the final value corresponds to the fill mode of **freeze**. An animation that has completed and has a fill mode of **freeze** is a frozen animation.<sup>2</sup>

An important note about a frozen animation is that while it might seem that an animation that is frozen will be frozen with the last value that is defined for the animation this is not the case. For example, if an animation is defined to interpolate a property from a value of zero to a value of ten and that animation is frozen after it completes, the frozen value is *not* necessarily ten. This is because it is possible for an animation to cease part way through its natural interval.<sup>3</sup> One common manner in which this situation might arise is if a non-integral **repeatCount** is specified. For example, a **repeatCount** of 0.5 is entirely valid, as is 3.23. In the first case the animation would stop half-way through its natural interval and may be frozen with the value of five.

Without going any deeper into the semantics of SMIL we summarise that ‘to animation’ produces a special combination of addition and interpolation; animations may be frozen after they complete; and the frozen value may represent a point part-way through the animation’s interpolation interval. Now we come to the complication with frozen ‘to animations’.

Generally if an animation is frozen it will continue to be combined with lower

---

<sup>2</sup>This description must be read with the concept of the animation sandwich in mind. The fact that an animation is frozen does not mean the target attribute is also constant as there may be another animation that is contributing to the final value of the target attribute that is still in progress.

<sup>3</sup>By natural interval I mean what SMIL terms the simple duration.

priority animations in the same manner as if it were still animating. However, this is not the case for ‘to animation’ which combines differently when frozen.

Following is an excerpt from the SMIL Animation specification. I have included the entire paragraph as the first half summarises what I have just attempted to explain. The second half describes how a ‘to animation’ is frozen.

Note that if no other (lower priority) animations are active or frozen, this [to animation] defines simple interpolation. However if another animation is manipulating the base value, the *to-animation* will add to the effect of the lower priority, but will dominate it as it nears the end of the simple duration, eventually overriding it completely. The value for  $F(t)$  when a to-animation is frozen (at the end of the simple duration) is just the to value. If a *to-animation* is frozen anywhere within the simple duration (e.g., using a repeatCount of “2.5”), the value for  $F(t)$  when the animation is frozen is the value computed for the end of the active duration. Even if other, lower priority animations are active while a to-animation is frozen, the value for  $F(t)$  does not change.

*Source:* SMIL Animation section 3.3.6, emphasis original.

In this extract we may consider  $F(t)$  to be the result of the ‘to animation’ after combining with lower priority animations but before combining with higher priority animations.<sup>4</sup>

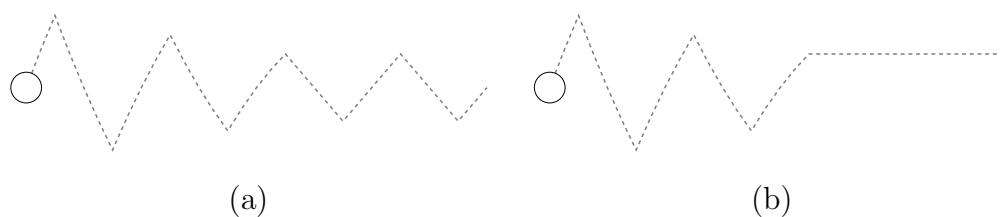
When the animation finishes at the end of its natural interval (called ‘simple duration’ in the above quotation) the result is as we would expect: the ‘to’ value. This is consistent with the definition of freezing for other kinds of animation. However, when an animation is frozen part-way through the natural interval it ceases to be combined with lower-priority animations and simply remains fixed at its final value. This is quite surprising.

---

<sup>4</sup>The actual definition is that  $F(t)$  represents the animation effect at time  $t$  at this point in the animation sandwich.



Consider the two animations shown in Figure 6.3. Each animation is based on freezing the animation previously described in Figure 6.2 part-way through its natural interval. On the left the path of the animation corresponds to the result if the ‘to animation’ continued to combine with the underlying value in the same manner as when it was active. On the right, the path of the animation corresponds to the behaviour defined in the SMIL Animation specification.



**Figure 6.3:** Alternative behaviour for a frozen ‘to animation’. The path shown in (a) corresponds to the behaviour that naturally falls out of the model and the behaviour of many SMIL implementations. The path shown in (b) corresponds to the behaviour defined by the SMIL Animation specification.

Producing the behaviour on the right requires that at each point after the ‘to animation’ is frozen, the underlying animations are sampled with a sample time corresponding to the sample time when the ‘to animation’ was frozen. Achieving this result is particularly complicated. In the design I have described where the timing and animation models are considered to be largely independent of one another, implementing this behaviour requires introducing coupling between the two models. The animation model must communicate to the timing model that it has a frozen ‘to animation’ and so the timed elements that correspond to underlying values of this ‘to animation’—of which the timing model has no notion—need to be sampled with a different sample time.

From this description it should be clear that implementing this correctly would compromise the design I have described as a great deal of special handling would be required for a situation that I expect only arises very rarely. This is perhaps why every implementation of SMIL Animation I have experimented with produces the behaviour corresponding to the animation on the left of Figure 6.3 rather than the right.

In this project I have produced an imperfect implementation of the behaviour described in the SMIL Animation specification that avoids introducing coupling between the timing and animation model. The approach I have taken is simply to store the final result of evaluating a ‘to animation’ after it first becomes frozen and then to use this result in subsequent samples. This approach will produce the correct behaviour provided the frame rate of the animation is sufficiently rapid. Just how rapid depends on the rate at which the animation changes but for the majority of cases any inaccuracies would be undetectable. The other situation in which this approach fails is when a ‘seek’ is performed on the animation causing it to jump to a time where a ‘to animation’ is frozen part-way through its natural interval and is combined with lower priority animations.

I do not mention this example as a criticism of the SMIL Animation specification although as an implementor I would prefer that the definition of a frozen ‘to animation’ were defined otherwise. My purpose in describing this situation is simply to give the reader an appreciation of some of the complexities involved in implementing SMIL Animation.

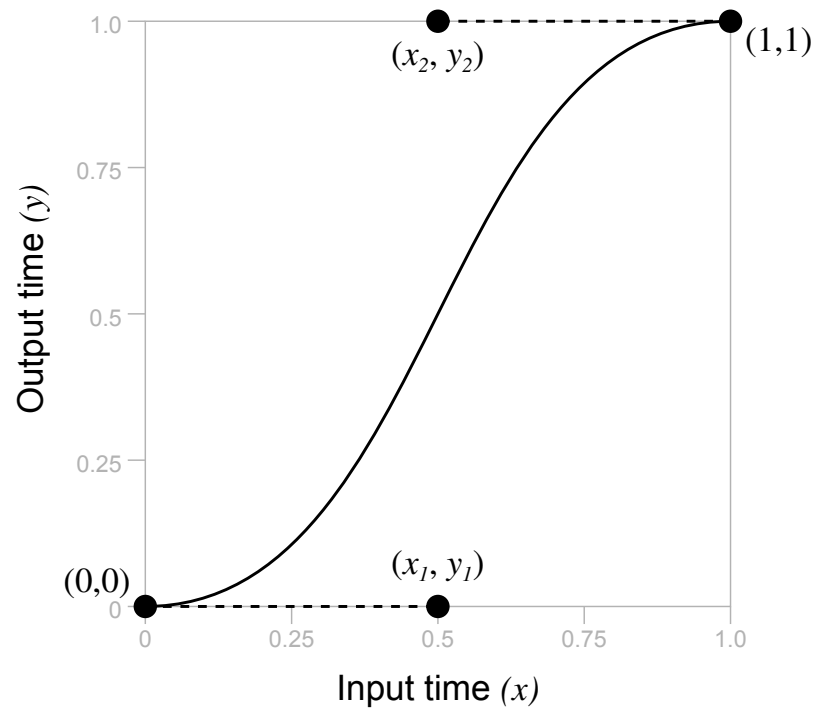
## 6.4 Spline-based timing

SMIL Animation includes a feature for controlling the timing of an animation by using a function defined using a cubic Bézier curve. This function takes time as an input parameter and scales the result to produce an adjusted output time. Figure 6.4 shows a graph of this function. The end points of the Bézier curve are fixed at  $(0, 0)$  and  $(1, 1)$  and the values of the two control points are set using the `keySplines` attribute. This can be used to produce acceleration and deceleration effects such as the popular ease-in, ease-out effect.

A cubic Bézier curve is described by the following equation (Shirley 2002, p. 234):

$$\mathbf{B}(t) = \mathbf{P}_0(1 - t)^3 + 3\mathbf{P}_1t(1 - t)^2 + 3\mathbf{P}_2t^2(1 - t) + \mathbf{P}_3t^3, \quad t \in [0, 1].$$

For a two dimensional curve on an  $x, y$  plane we can also express this as



**Figure 6.4:** The spline graph used by the `keySplines` attribute.

(Lancaster 1998):

$$\begin{aligned}x(t) &= At^3 + Bt^2 + Ct + D \\y(t) &= Et^3 + Ft^2 + Gt + H\end{aligned}$$

where:

$$\begin{aligned}A &= x_3 - 3x_2 + 3x_1 - x_0 & E &= y_3 - 3y_2 + 3y_1 - y_0 \\B &= 3x_2 - 6x_1 + 3x_0 & F &= 3y_2 - 6y_1 + 3y_0 \\C &= 3x_1 - 3x_0 & G &= 3y_1 - 3y_0 \\D &= x_0 & H &= y_0\end{aligned}$$

In these equations  $(x_0, y_0)$  and  $(x_3, y_3)$  are the endpoints of the curve and  $(x_1, y_1)$  and  $(x_2, y_2)$  are the control points that define the shape of the curve between the endpoints. For the graph used in SMIL Animation the end points are fixed at  $(0, 0)$  and  $(1, 1)$  which simplifies the equations somewhat.

The difficulty with these definitions however is that they are expressed in terms of the parameter  $t$ .<sup>5</sup> In calculating a scaled time value however we require the  $y$  value corresponding to a given  $x$  representing the input time. This is a common problem and several solutions have been proposed which I will describe.

The first solution that I applied was the suggestion put forward by Lancaster (1998). This approach begins by using the  $x$  value as an initial guess for  $t$  and then using Newton-Raphson iteration to converge on a root.

Newton-Raphson iteration is a root-finding technique that uses straight line segments created at a tangent to the curve from the current approximate root to find a new root where the tangent crosses the  $x$  axis (Hearn & Baker 1997, pp. 621–622). This technique is popular in graphics programming because if it converges on a root it will do so faster than any other technique (Hearn & Baker 1997, p. 622).

In order to create a tangent to the curve the slope of the curve must be calculated.

---

<sup>5</sup>The naming of variables is unfortunate in this case. I have chosen to use  $t$  as the parameter to the Bézier equation in keeping with the literature and  $x$  to represent the input time and  $y$  for the scaled output time.

For cubic Bézier curves this is simply:

$$x'(t) = 3At^2 + 2Bt + C$$

The next root is simply the point where this tangent crosses the  $x$  axis (Hearn & Baker 1997, p. 622):

$$t_1 = t_0 - \frac{x(t_0)}{x'(t_0)}$$

This procedure is repeated iteratively until the desired accuracy is achieved. Lancaster (1998) suggests that three iterations are usually sufficient. After a root has been found, this value for  $t$  is used as input to the Bézier function for  $y$  to produce the scaled result.

This approach has the benefit that it is simple and fast as each iteration requires little calculation. However, in my testing I discovered that in some cases three iterations was insufficient. For example, the case where the first control point is at  $(1, 1)$  and the second at  $(0, 0)$  produces wild results after only three iterations. In this case at least fifteen iterations are needed to produce a sensible result.

Watt & Policarpo (2001, pp. 373–375) tackle a similar re-parameterisation problem. The case they consider is for a function that maps time to velocity whereas for SMIL Animation our function maps input time to output time. Nevertheless Watt & Policarpo's approach is useful. They suggest building up a table of cumulative chord lengths. For SMIL the same can be done only here it is much easier. We simply build up a table of sample values equally spaced along the Bézier curve. Then when we come to calculate a value we first find the closest matching value in the sample table, perform some simple interpolation and then use this value as our initial guess for the Newton-Raphson iteration. Using this approach and a table of twenty values the function converges to a high degree of accuracy within three iterations. For a smaller table of ten values, four iterations are required for the same accuracy. We exploit the fact that the function is strictly monotonically increasing to perform the lookup.

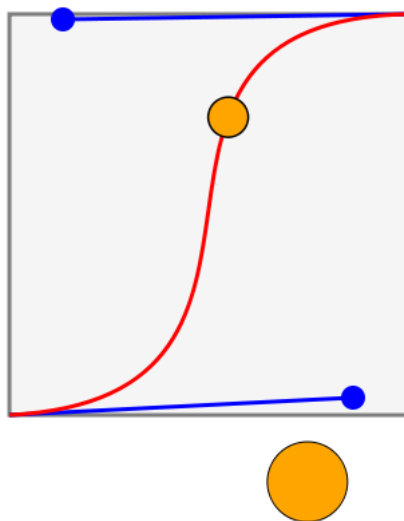
This table needs only be built once for each spline and can be re-used for every

calculation thereafter. The resulting calculation time for each frame is minimal. For the case when  $(x_1, y_1)$  and  $(x_2, y_2)$  define a straight line no calculation is performed but the value is returned unscaled. Further optimisations could be performed such as performing a binary search to find the nearest matching value in the table but such optimisations are not likely to produce a significant benefit, especially compared to the amount of time required for rendering.

Parent (2001, pp. 83–84) suggests a similar approach but using binary subdivision and only when necessary. This approach may be superior for more complex curves or when there are a large number of curves. However, for SMIL Animation the number of spline definitions is typically small, and they are typically encountered during document parsing so the performance penalty is paid up-front and is nevertheless minimal.

The memory penalty is more significant. For a lookup table of eleven 64-bit floating point numbers (as is the current implementation) 88 bytes are required. For an atypically complex animation with 100 spline intervals 8,800 bytes are required. This is not dramatic, particularly when compared to the size of the other components in the document. However, if this memory requirement did prove prohibitive we could easily half the size of the table and increase the number of iterations but for such a complex animation speed requirements are probably more critical and this approach is justified.

In order to test this function I have produced the simple SVG graphic shown in Figure 6.5. This graphic allows the user to manipulate a cubic Bézier curve interactively which will then be used as input to a motion animation. A second object located at the origin of the graph is animated by two functions. On the horizontal axis a constant animation moves the object to the right whilst on the vertical axis the function defined by the curve is applied to the object. Under this arrangement the object follows the curve defined by the user. The idea for this test case is based on a similar tool by Lussier (n.d.).



**Figure 6.5:** Screenshot from the `keySplines` test graphic.

## 6.5 Ownership

The lifetimes of XPCOM components are managed by a reference counting scheme. Therefore, in order to prevent memory leaks or dangling pointers it is necessary to consider ownership relationships carefully. The most common problem is to define a circular ownership relationship in which memory is never released.

I have incorporated my analysis of these relationships into the class diagrams in Chapter 5. In these diagrams composition (the filled diamond) indicates exclusive ownership where the owned class's lifetime depends solely on its owner. In XPCOM terms this implies that the owner is the only object holding a prolonged owning reference to the owned object although other objects may have weak references to it.

Aggregation (the hollow diamond) indicates shared ownership which in XPCOM terms implies that several objects have owning references to the owned object.

Association (the unadorned line with an arrow) indicates a weak relationship which is expressed in XPCOM as a weak reference or a short-lived owning refer-

ence although in fact these two are identical.

Based on this notation we can identify potential ownership cycles by attempting to draw circles on the diagrams that follow only composition or aggregation relationships and only in the direction of ownership. Such circles do not always reveal ownership cycles as, for example, the objects acting as ‘owner’ and ‘owned’ may be different objects of the same class. There are other situations in which a cycle may be intended but this technique nevertheless assisted me in defining ownership relationships and identifying problem areas.

In addition to this I applied some of the memory leak detection tools developed for tracking XPCOM memory leaks. These tools identified a particularly harmful leak that occurred at each animation sample.

## 6.6 Performance tuning

At the completion of the second iteration of development I conducted some performance analysis. Without any profiling tools I was limited to using some simple timing macros provided along with the Mozilla codebase. Using this simple technique I determined that 93% of the time spent in each animation sample was spent in rendering. Of the remaining 7% the largest single step at about 40~45% was setting the value inside the SVG model. My approach was therefore to reduce the amount of rendering performed which I achieved using a simple optimisation.

If the parameters supplied to the animation function do not vary between two samples then neither will the output. Based on this observation I created a simple flag that is set whenever one of the animation function’s parameters are changed and is reset whenever the animation function is evaluated. This flag is accessible via the composable interface so that the compositor can quickly determine if any of the animations targeting a given attribute have changed.

A second step in this optimisation is based on the fact that some animations overwrite underlying values. In this case it is not necessary to calculate the underlying value. By exposing this information to the compositor these overwritten



animations can be filtered out so that they are not evaluated. This is a minor performance win as these functions are cheap to evaluate compared to the render time. However, this second analysis step prevents animations that will be overwritten but which have changed parameters from causing the attribute to be updated.

Using these optimisation if no animations have changed then the target attribute does not need to be updated. If no target attributes are updated—as is often the case such as when all animations are frozen—then the graphic is not redrawn. But even if only a few target attributes are not updated there is still a performance gain due to this optimisation as the dirty region of the image that needs to be redrawn may be smaller.

Following these optimisations I attempted to compare the results against the baseline but was unable to record sensible results from all the test cases. For those cases for which I could record results, the time spent in the animation loop after introducing the optimisations was about 50% less than before introducing the optimisations. When no animations were active, the time spent in the loop was nearly 100% less which is to be expected.

## 6.7 Threading

My initial understanding of the Mozilla codebase lead me to expect that there was potential for race conditions to arise as the timer thread performing the animation may access the same parts of the model as the thread responsible for executing scripts. As a result I implemented locking for these parts of the model. I have since been informed that due to the implementation of timers and script execution this situation should not arise. As a result the part of the code that deals with locking is not necessary and will be removed.

# Chapter 7

## Conclusions

On 4<sup>th</sup> November 2005 I submitted a patch to the Mozilla defect tracking system containing my implementation of SMIL Animation. This chapter describes the contents of this patch and my concluding thoughts on the future of this work and the associated technologies.

### 7.1 Implementation status

Table 7.1 outlines the features of SMIL Animation that I have successfully implemented in this project. This feature set corresponds to a basic SMIL model as was my intention in undertaking this project. The model is basic in that some of the more complex features such as synchbase timing and event-based timing are not implemented. However, nearly all the fundamental timing and animation properties are fully implemented. The only missing feature in this area is `calcMode` support which is partially implemented and the remaining work is minimal.

I describe the implemented feature set as a ‘model’ because the area in which this code is most lacking is in the bindings to SVG data-types. As I have described in Chapter 6 this is due to ongoing discussions in the Mozilla developer community and is outside of my control.

**Table 7.1:** SMIL Animation features implemented in this project.

Feature	Status
<code>&lt;animate&gt;</code>	
<code>xlink:href</code>	Not implemented
<code>attributeName</code>	Implemented
<code>attributeType</code>	Not implemented
<code>begin, end</code>	
<code>offset values</code>	Implemented
<code>syncbase values</code>	Not implemented
<code>event values</code>	Not implemented
<code>repeat values</code>	Not implemented
<code>accessKey values</code>	Not implemented
<code>wallclock values</code>	Not implemented
<code>dur</code>	Implemented
<code>min, max</code>	Implemented
<code>restart</code>	Implemented
<code>repeatCount</code>	Implemented
<code>repeatDur</code>	Implemented
<code>fill</code>	Implemented
<code>calcMode</code>	Linear and spline modes only
<code>values</code>	Implemented
<code>keyTimes</code>	Implemented
<code>keySplines</code>	Implemented
<code>from</code>	Implemented
<code>to</code>	Implemented
<code>by</code>	Implemented
<code>additive</code>	Implemented
<code>accumulate</code>	Implemented
<code>&lt;set&gt;</code>	Not implemented
<code>&lt;animateMotion&gt;</code>	Not implemented

Table 7.1 (continued)

Feature	Status
<animateColor>	Not implemented
<animateTransform>	Not implemented
Interface ElementTimeControl	Implemented
Interface TimeEvent	Not implemented
Interface SVGAnimationElement	Not implemented
SVGSVGElement animation related DOM methods	
pauseAnimations	Implemented
unpauseAnimations	Implemented
animationsPaused	Implemented
getCurrentTime	Implemented
setCurrentTime	Not implemented
dur	Implemented
Related features	
Animation of SVG types	Not implemented (a basic implementation for <code>SVGAnimatedLength</code> 's is provided for testing however)
Animation of CSS properties	Not implemented
Proper handling of relative values	Not implemented

At the time of writing the feedback I have received to this work has been positive and it is my expectation that the code produced will ultimately become part of the Mozilla codebase. Before that occurs however, this code will undergo the review process that is common to all contributions to Mozilla and this will undoubtedly involve changes to the code before it is accepted.

Once this code becomes part of the Mozilla codebase it will be much easier for other developers to contribute to producing a complete SMIL implementation. It my hope that this work will be complete in time for the next major revision of the Firefox Web browser due in 2006. Because of extensive the distribution of

Mozilla Firefox this would represent a significant step toward providing a more sophisticated Web application platform by extending the number of users able to view SMIL animated content by hundreds of millions.

The realisation of this goal however may be delayed due to rework of the Mozilla layout engine and may be complicated if incompatibilities between SMIL implementations are not resolved. Fortunately some individuals have taken on the task of testing competing implementations and documenting their discrepancies. These efforts deserve wider recognition and may benefit from pooling their test cases in a coordinated manner.

In the interim period Schepers (2005) has developed SmilScript. SmilScript is an implementation of SMIL Animation using script. This allows SVG implementations that do not yet support SMIL to run a script which will provide the SMIL behaviour. As Schepers acknowledges, this approach has several limitations. In particular, because the implementation is script-based proper separation of animation and base values is not possible and the performance is limited. Schepers acknowledges that this is only a temporary measure but nevertheless it is a very useful measure whilst SMIL in SVG is not widely implemented and may encourage SVG authors to experiment with this technology.

As for this project, I consider that it has achieved all the targets set. The goal of developing a basic model, integrated with the codebase has been reached as outlined above; the performance is adequate as described in the previous chapter; and the risks identified have not eventuated. Furthermore, I have produced documentation to assist developers to continue this work and have begun the process that I expect will ultimately result in this code being accepted into the target codebase.

## 7.2 Future directions

In his keynote address at the SVG Open 2005 Conference, Cagle (2005) acknowledged that, “Open standards is a powerful concept, and it works against the

status quo”. For some companies, SVG may represent a threat to their control of a particular market and instead may push them to “compete on that one quality that most companies prefer not to compete on—quality”. This observation picks up on an unavoidable fact about the future success of SVG and indeed the Web platform as a whole: it will not be decided based on technical merit alone.

Many have speculated on the implications of Adobe—producer of the most widely used SVG viewer to date—merging with Macromedia—producer of Flash. The result of this move may become a turning point for SVG, for better or worse. Similarly Microsoft’s XAML technology represents another technology that will certainly influence the industry even if only because it is backed by Microsoft.

It is possible to extend speculations even to the political domain. For example, if diplomatic relations between the United States and emerging Asian nations such as China prevent companies such as Microsoft from gaining a strong position in these markets then this would certainly affect the adoption and popularity of open alternatives such as those technologies that make up the Web platform.

Whatever the future holds, the Web platform already enables millions of applications from banking to email. It provides users with greater physical freedom and allows computer applications to be extended to wider audiences as it does not discriminate against specific hardware or software platforms. Yet it is worth pausing at this moment to consider the significance of this development. Technological progress should not remain unchecked.

Clearly the Web platform can provide benefits to many but it can equally be said that in an era where information is regarded as a corporation’s most valuable asset the Web platform places more information in the hands of those who own the applications. Increasingly, individuals are at the mercy of corporations and governmental regulations to ensure that their information is not used unscrupulously.

Similarly, by enabling applications to be moved from the desktops of customer service assistants to customers’ living rooms, individuals may be expected to interact with computers rather than people. There are two obvious implications of this trend.

Firstly, such a transition threatens the jobs of those who would otherwise serve customers personally. The shrinking of various sectors of employment has accompanied nearly every technological change, but as Mesthene (1970, p. 27) points out there is no evidence to suggest that this will ultimately deprive the population of work. Rather, using the example of the introduction of the automobile, Mesthene describes how although some jobs were eliminated, the total number of jobs in the US did not decline as a result of this new technology but rather increased. In the same way the Web platform may cause some workers to be displaced but it unlikely to lead to large-scale unemployment.

Another implication of making individuals the direct users of corporate or government applications that many have noted is the possibility of creating a new lower class based on technological competence and access to technology: the well-documented ‘digital divide’. The complementary situation where the technologically-savvy control society—technocracy—has clearly not eventuated (Elsner 1967, p. 186) but this less confronting situation may have already arrived. As a result the International Telecommunication Union (ITU), an agency of the United Nations have established the World Summit of the Information Society to tackle these issues. The results of this summit are yet to become clear but the very establishment of the summit confirms a widening awareness of this situation. It is confirmed by statistical measures (Norris 2001) as well as anecdotal evidence such as the struggles experienced by the elderly in attempting to use automatic teller machines (Williams 1997). At the current time it seems that the only factors preventing a technology-driven social division from being further exaggerated are governmental regulations and the conscience of various advocates and organisations.

Another less obvious implication of this change is that computer applications typically provide fewer avenues for redress in the case of a mistake or exceptional circumstances when compared to interacting with another person. The net effect is that unless specific measures are taken to ensure these situations can be properly handled individuals may be disempowered or, more subtly, the values of our society may shift.

Mesthene (1970, p. 50) asserts that technology that can bring about changes in

social values by altering the range of choices available and the costs associated with them. If the cost of fixing computer mistakes is too great what will be the effect on the value we place on accurate (as opposed to precise) information—on truth? Or if the effort required to accommodate situations beyond the programming of computer applications is too great, what will be the result for our notions of giving proper treatment to all people—to fairness and justice? Or will we become less flexible like the computer programs we interact with?

These are interesting topics, but they go well beyond this project. For now we note that these concerns are valid and deserve the attention being brought to bear on them by various advocates. Nevertheless I consider that SVG has at least three redeeming qualities in this regard.

Because SVG is an open standard it does not discriminate against any particular implementation or any particular hardware or software. It is therefore possible for SVG to be realised on lower-cost platforms which in turn lowers the barrier for entry that might otherwise exclude impoverished peoples and nations from participating in the technological market.

A second quality of SVG that may be argued in its defence against the criticisms I have outlined is that it enables the development of more intuitive user interfaces. Declarative animation as I have implemented could be used to provide visual cues to a user who is unfamiliar with computers as to what interaction is expected. This quality may be make computers more accessible for those with disabilities or with limited computer skills.

Finally, SVG is a client-side technology. By shifting more interaction to the client-side, less information needs to be exchanged with a central server which may mitigate privacy issues, particularly as some applications may be able to be run in an offline mode.

For these reasons I believe SVG as a technology can at least be said to provide as much potential for charity as it does for harm. It is my hope that through this Capstone project I have contributed to realising a richer, more intuitive platform for future technological applications and in a manner that will benefit most.



# Glossary

**AJAX** Asynchronous JavaScript and XML. A combination of client-side Web technologies that allows a more interactive communication pattern between a Web browser and Web server.

**API** Application Programming Interface. The protocol by which an application communicates with another software component.

**ARPANET** Advanced Research Projects Agency Network. A project of the U.S. Department of Defence that formed the basis for much of the technology that is now at the core of the Internet architecture.

**CSS** Cascading Style Sheets. A standard for describing formatting information to be applied to a document, typically an HTML Web page.

**DOM** Document Object Model. An object-oriented API for manipulating an XML document.

**DTD** Document Type Definition. The formal definition of a particular class of XML or SGML documents.

**ECMAScript** A standardised version of the JavaScript programming language.

**Firefox** A popular Web browser produced by the Mozilla project.

**Flash** A technology owned by Macromedia commonly used on Web pages to produce interactive graphics.

**HTML** Hypertext Markup Language. The format used for describing nearly every Web page on the Internet.

**JavaScript** A scripting language developed by Netscape that is commonly used for client-side Web development.

**MMS** Multimedia Message Service. A system used to deliver messages over wireless networks (typically to mobile phones) that may include a multimedia component.

**MySQL** A popular open-source database often used in Web development projects along with a scripting language such as PHP.

**PHP** PHP: Hypertext Processor. A popular scripting language commonly used for server-side Web development.

**PostgreSQL** An advanced open-source database used in many Web development projects.

**Python** A popular scripting language used in many areas including Web development.

**RTTI** Run-Time Type Information. A feature of the C++ language that allows the type of objects to be determined during program execution.

**Ruby** A markup language for annotating text such as is commonly found in Japanese writing.

**SGML** Standard Generalised Markup Language. A text format for representing structured information that is a predecessor to XML.

**SMIL** Synchronised Multimedia Integration Language. An XML-based language used to describe elements in an interactive audiovisual display.

**SVG** Scalable Vector Graphics. An XML-based language for describing vector graphics.

**W3C** World Wide Web Consortium. The standards body responsible for many of the technology standards in use on the Web.

**XHTML** A successor to HTML that uses XML as an underlying format.

**XML** Extensible Markup Language. A common data format for representing structured information upon which many standards are based.

**XPCOM** Cross-Platform Component Object Module. A component technology developed by and used in the Mozilla project.

**XSLT** Extensible Stylesheet Language Transformations. An XML-based language for transforming an XML data source into a (typically XML-based) alternative form.

**XUL** XML User Interface Language. A markup language developed by the Mozilla project for describing the components of a user interface.

# References

- Akroyd, M. (1996), ‘Antipatterns : Vaccinations against object misuse’, *Object World West, San Francisco* .
- Aulenback, S. (2004), SVG as the visual interface to Web services, *in* V. Geroimenko & C. Chen, eds, ‘Visualizing Information Using SVG and X3D’, Springer, chapter 4, pp. 85–98.
- Beck, K. (2000), *Extreme programming explained : embrace change*, Addison-Wesley, New Jersey.
- Beck, K., Beedle, M., van Bennekum, A., Cockburn, A., Cunningham, W., Fowler, M., Grenning, J., Highsmith, J., Hunt, A., Jeffries, R., Kern, J., Marick, B., Martin, R. C., Mellor, S., Schwaber, K., Sutherland, J. & Thomas, D. (2001), ‘The agile manifesto’.  
Available at: <http://agilemanifesto.org/>
- Bell, G. (1999), The folly of prediction, *in* P. J. Denning, ed., ‘Talking Back to the Machine : Computers and Human Aspiration’, Springer, chapter 1, pp. 1–14.
- Berners-Lee, T. (1990), ‘WorldWideWeb - summary’.  
Available at: <http://www.w3.org/History/19921103-hypertext/hypertext/WWW/Summary.html>
- Berners-Lee, T., Hendler, J. & Lassila, O. (2001), ‘The Semantic Web’, *Scientific American* **284**(5), 34–43.

- Boehm, B. (2000), *Spiral development : Experience, principles, and refinements*, Special Report CMU/SEI-00-SR-08, The Software Engineering Institute.
- Bucken, M. W. (2003), 'Client-side Java ain't dead yet', *Application Development Trends magazine* .  
Available at: <http://www.adtmag.com/article.asp?id=7081>
- Bunker, D. (n.d.), 'Why to avoid Flash'.  
Available at: [http://people.westminstercollege.edu/students/d-b1649/opinions/avoid\\_flash.html](http://people.westminstercollege.edu/students/d-b1649/opinions/avoid_flash.html)
- Cagle, K. (2004), *Distributed user interfaces : Toward SVG 1.2*, in V. Geroimenko & C. Chen, eds, 'Visualizing Information Using SVG and X3D', Springer, chapter 6, pp. 119–152.
- Cagle, K. (2005), 'The future of SVG and the Web', *Understanding XML* .  
Available at: [http://www.understandingxml.com/archives/2005/08/the\\_future\\_of\\_s.html](http://www.understandingxml.com/archives/2005/08/the_future_of_s.html)
- Collins, S. (2002), 'C++ portability'.  
Available at: [http://www.mozilla.org/hacking/cpp\\_portability/](http://www.mozilla.org/hacking/cpp_portability/)
- Deakin, N. (2003), 'Building technologies with baby steps'.  
Available at: <http://www.xulplanet.com/ndeakin/article/215/>
- Dsouza, A., Hildebrand, K. & Israeli, G. (2004), 'Conceptual architecture of Mozilla'.  
Available at: <http://www.cs.uwaterloo.ca/~kdhildeb/cs746/conceptual.pdf>
- Elsner, Jr., H. (1967), *The Technocrats : Prophets of automation*, Syracuse University Press.
- Fowler, M. (1999), *Refactoring : Improving the design of existing code*, Addison-Wesley, Massachusetts.
- Futrell, R. T., Shafer, D. F. & Shafer, L. I. (2002), *Quality software project management*, Prentice Hall.

- Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1995), *Design patterns : Elements of re-usable object-oriented software*, Addison-Wesley.
- Garrett, J. J. (2005), 'Ajax : a new approach to web applications'.  
Available at: <http://www.adaptivepath.com/publications/essays/archives/000385.php>
- Granneman, S. (2005), *Don't Click on the Blue E!*, O'Reilly.
- Hearn, D. & Baker, M. P. (1997), *Computer graphics : C version*, 2nd edn, Prentice Hall.
- Himanen, P., Torvalds, L. & Castells, M. (2002), *The Hacker Ethic*, Random House.
- Hoffmann, O. (2005), 'Examples and tests for SVG animation'.  
Available at: <http://olaf.s02.user-portal.com/svgtest/>
- King, P., Schmitz, P. & Thompson, S. (2004), Behavioral reactivity and real time programming in XML : Functional programming meets SMIL animation, in 'DocEng '04: Proceedings of the 2004 ACM symposium on Document engineering', ACM Press, New York, NY, USA, pp. 57–66.
- Lancaster, D. (1998), 'More on cubic spline math'.  
Available at: <http://www.tinaja.com/text/bezmath.html>
- Lussier, R. K. (n.d.), 'Key splines graph tool'.  
Available at: <http://www.burningpixel.com/svg/keySplineTool.htm>
- Mackinnon, T., Freeman, S. & Craig, P. (2000), 'Endo-testing: Unit testing with mock objects'.  
Available at: <http://www.connextra.com/aboutUs/mockobjects.pdf>
- Markham, G. (2005), 'XAML and XUL'.  
Available at: [http://weblogs.mozillazine.org/gerv/archives/2005/05/xaml\\_and\\_xul.html](http://weblogs.mozillazine.org/gerv/archives/2005/05/xaml_and_xul.html)

- Marson, I. (2005), 'Firefox : the alternative history', *ZDNet UK* .  
Available at: <http://insight.zdnet.co.uk/software/applications/0,39020466,39208866,00.htm>
- McFarlane, N. (2003a), 'Longhorn and Mozilla : Birds of a feather', *DevX.com* .  
Available at: <http://www.devx.com/DevX/Article/17899>
- McFarlane, N. (2003b), *Rapid Application Development with Mozilla*, Prentice Hall.
- Mesthene, E. G. (1970), *Technological change : Its impact on man and society*, Harvard University Press.
- mozilla.org (2000), 'About NSPR'.  
Available at: <http://www.mozilla.org/projects/nspr/about-nspr.html>
- mozilla.org (2005a), 'Good first bugs'.  
Available at: <http://www.mozilla.org/contribute/hacking/first-bugs/>
- mozilla.org (2005b), 'Mozilla foundation reorganization'.  
Available at: <http://www.mozilla.org/reorganization/>
- Neumann, A. & Winter, A. M. (2004), 'Comparing .SWF (ShockWave Flash) and .SVG (Scalable Vector Graphics) file format specifications'.  
Available at: [http://www.carto.net/papers/svg/comparison\\_flash\\_svg/](http://www.carto.net/papers/svg/comparison_flash_svg/)
- Norris, P. (2001), *Digital Divide : Civic engagement, information poverty, and the Internet worldwide*, Cambridge University Press.
- Oeschger, I., Murphy, E., King, B., Collins, P. & Boswell, D. (2002), *Creating applications with Mozilla*, O'Reilly.
- Parent, R. (2001), *Computer Animation*, Morgan Kaufmann.
- Parisi, T. (2004), Correcting the great mistake, in V. Geroimenko & C. Chen, eds, 'Visualizing Information Using SVG and X3D', Springer, chapter Foreword, pp. vii–viii.

- Rowley, T., Morris, J., Watt, J. & Fritze, A. (2005), 'Implementing SVG in a web browser : Past, present, and future of Mozilla SVG', *SVG Open 2005* . Available at: <http://www.svgopen.org/2005/papers/MozillaSVG/>
- Schepers, D. (2005), 'SmilScript'. Available at: <http://www.vectoreal.com/smilscript/>
- Schmidt, D. & Stephenson, P. (1994), Achieving reuse through design patterns : A case study of evolving object-oriented system software across OS platforms, in 'Proceedings of the 3rd SIGS C++ World Conference 1994'. Available at: <http://www.cs.wustl.edu/~schmidt/PDF/C++-world-94.pdf>
- Schmitz, P. L. (n.d.), 'Timing and animation support for Batik : Detailed design document'. Available at: <http://www.ludicrum.org/plsWork/papers/BatikSMILsupport.htm>
- Shirley, P. (2002), *Fundamentals of computer graphics*, A K Peters.
- svg.org (2005), 'Shipping and announced SVG phones'. Available at: [http://svg.org/special/svg\\_phones](http://svg.org/special/svg_phones)
- Torvalds, L. (1999), The Linux edge, in C. DiBona, S. Ockman & M. Stone, eds, 'Open Sources', O'Reilly, pp. 101–112.
- WaSP (n.d.), 'The Acid2 browser test'. Available at: <http://www.webstandards.org/act/acid2/>
- Watt, A. & Policarpo, F. (2001), *3D Games : Real-time rendering and software technology*, Vol. One, Addison-Wesley.
- WHATWG (2004), 'Web Hypertext Application Technology Working Group charter'. Available at: <http://www.whatwg.org/charter>
- WHATWG (2005), 'Web Applications 1.0'. Available at: <http://www.whatwg.org/specs/web-apps/current-work/>



- Williams, D. (1998), ‘C++ portability guide’, *mozilla.org* .  
Available at: <http://www.mozilla.org/hacking/portable-cpp.html>
- Williams, P. (1997), ‘Elderly have problems with bank ATMs’, *University of Georgia Campus News* .  
Available at: <http://www.uga.edu/columns/112497/camp5.html>
- World Wide Web Consortium (2001), ‘SMIL Animation’.  
Available at: <http://www.w3.org/TR/smil-animation>
- World Wide Web Consortium (2004), ‘The W3C workshop on Web applications and compound documents’.  
Available at: <http://www.w3.org/2004/04/webapps-cdf-ws/>
- World Wide Web Consortium (2005a), Compound document by reference use cases and requirements version 1.0, Working draft, World Wide Web Consortium.  
Available at: <http://www.w3.org/TR/2005/WD-CDRReq-20050809/>
- World Wide Web Consortium (2005b), Compound Document Framework 1.0 and WICD Profiles, Working Draft, World Wide Web Consortium.  
Available at: <http://www.w3.org/TR/2005/WD-CDRReq-20050809/>

# Appendix A

## Requirements database

Table A.1 lists the requirements in the database created for this project. The database also includes fields that contain a description of each requirement, comments, specification URLs, criticality and the iteration where the requirement is scheduled to be implemented. As it would be difficult to present all the data in this report I have included only a summary of the database here.

**Table A.1:** Requirements database

<b>ID</b>	<b>Title</b>	<b>Source</b>	<b>Section</b>
6	Correctly locates target elements using xlink:href	SVG 1.1	19.2.2, 19.2.4
7	document begin coincides with the SVGLoad event of the outermost svg element	SVG 1.1	19.2.2
8	All animations in an SVG document fragment must stop in the event of any error within the document	SVG 1.1	19.2.2
9	If no xlink:href is provided, the immediate parent element is the animation target	SVG 1.1	19.2.4
13	attributeName attribute	SVG 1.1	19.2.5

Table A.1 (continued)

<b>ID</b>	<b>Title</b>	<b>Source</b>	<b>Section</b>
14	attributeType attribute	SVG 1.1	19.2.5
15	begin attribute	SVG 1.1	19.2.6
16	begin: offset values	SVG 1.1	19.2.6
17	begin: syncbase	SVG 1.1	19.2.6
18	begin: event values	SVG 1.1	19.2.6
19	begin: repeat value	SVG 1.1	19.2.6
20	begin: access key	SVG 1.1	19.2.6
21	begin: wallclock	SVG 1.1	19.2.6
22	begin: indefinite	SVG 1.1	19.2.6
23	dur attribute	SVG 1.1	19.2.6
24	dur: clock value	SVG 1.1	19.2.6
25	dur: media	SVG 1.1	19.2.6
26	dur: indefinite	SVG 1.1	19.2.6
27	end attribute	SVG 1.1	19.2.6
28	end: offset	SVG 1.1	19.2.6
29	end: syncbase	SVG 1.1	19.2.6
30	end: event	SVG 1.1	19.2.6
31	end: repeat	SVG 1.1	19.2.6
32	end: access key	SVG 1.1	19.2.6
33	end: wallclock	SVG 1.1	19.2.6
34	end: indefinite	SVG 1.1	19.2.6
35	min attribute	SVG 1.1	19.2.6
36	max attribute	SVG 1.1	19.2.6
37	restart attribute	SVG 1.1	19.2.6
38	restart: always	SVG 1.1	19.2.6
39	restart: whenNotActive	SVG 1.1	19.2.6
40	restart: never	SVG 1.1	19.2.6
41	repeatCount attribute	SVG 1.1	19.2.6

Table A.1 (continued)

<b>ID</b>	<b>Title</b>	<b>Source</b>	<b>Section</b>
42	repeatDur attribute	SVG 1.1	19.2.6
43	fill: remove	SVG 1.1	19.2.6
44	fill: freeze	SVG 1.1	19.2.6
45	fill attribute	SVG 1.1	19.2.6
46	Clock values syntax	SVG 1.1	19.2.6
47	calcMode attribute	SVG 1.1	19.2.7
48	calcMode: discrete	SVG 1.1	19.2.7
49	calcMode: linear	SVG 1.1	19.2.7
50	calcMode: paced	SVG 1.1	19.2.7
51	calcMode: spline	SVG 1.1	19.2.7
52	values attribute	SVG 1.1	19.2.7
53	keyTimes attribute	SVG 1.1	19.2.7
54	keySplines attribute	SVG 1.1	19.2.7
55	from attribute	SVG 1.1	19.2.7
56	to attribute	SVG 1.1	19.2.7
57	by attribute	SVG 1.1	19.2.7
58	Animation values must be legal	SVG 1.1	19.2.7
59	additive attribute	SVG 1.1	19.2.8
60	additive: sum	SVG 1.1	19.2.8
61	additive: replace	SVG 1.1	19.2.8
62	accumulate attribute	SVG 1.1	19.2.8
63	accumulate: sum	SVG 1.1	19.2.8
64	accumulate: none	SVG 1.1	19.2.8
65	Animation should inherit	SVG 1.1	19.2.9
66	animate element	SVG 1.1	19.2.10
67	set element	SVG 1.1	19.2.11
68	animateMotion element	SVG 1.1	19.2.12
69	animateColor element	SVG 1.1	19.2.13

Table A.1 (continued)

ID	Title	Source	Section
70	animateTransform element	SVG 1.1	19.2.14
71	Animation restrictions must be enforced	SVG 1.1	19.2.15
72	Only certain elements can be animated by animateMotion	SVG 1.1	19.2.15
73	Attributes that are not animatable cannot be the target of an animation	SVG 1.1	19.2.15
74	Support for additive animation depends on the syntax used.	SVG 1.1	19.2.15
75	Certain data types are additive	SVG 1.1	19.2.15
76	Certain data types can be targeted by animate elements	SVG 1.1	19.2.15
77	Certain data types can be targeted by set elements	SVG 1.1	19.2.15
78	Certain data types can be targeted by animateColor elements	SVG 1.1	19.2.15
79	Only transform-lists can be targeted by animateTransform elements	SVG 1.1	19.2.15
80	Additive animation	SVG 1.1	19.2.8
81	DOM interfaces	SVG 1.1	19.5
82	Interface ElementTimeControl	SVG 1.1	19.5
83	Interface TimeEvent	SVG 1.1	19.5
84	Interface SVGAnimationElement	SVG 1.1	19.5
85	Interface SVGAnimateElement	SVG 1.1	19.5
86	Interface SVGSetElement	SVG 1.1	19.5
87	Interface SVGAnimateMotionElement	SVG 1.1	19.5
88	Interface SVGMPathElement	SVG 1.1	19.5

Table A.1 (continued)

ID	Title	Source	Section
89	Interface SVGAnimateColorElement	SVG 1.1	19.5
90	Interface SVGAnimateTransformElement	SVG 1.1	19.5
91	Provide standard DOM2 access to animation attributes	SVG 1.1	19.5
92	from-to-by animation	SMIL Animation	3.2.2
93	Errors should be handled correctly	SVG 1.1	F.2
94	Error processing shall only occur when the presentation is updated	SVG 1.1	F.2
95	Error processing shall not occur while redraw is suspended	SVG 1.1	F.2
96	Animations should stop when errors occur	SVG 1.1	F.2
97	Helpful error messages should be presented	SVG 1.1	F.2
98	Errors caused by animations should be detected	SVG 1.1	F.2
99	Implement SVGSVGElement animation interfaces	SVG 1.1	5.17
100	Animation should affect the CSS cascade	SMIL Animation	3.5
101	Out of range values should be clamped as late as possible	SMIL Animation	3.5
102	Changes via the DOM must be reflected in the animation	SMIL Animation	3.5
103	Timing model must be end-point exclusive	SMIL Animation	3.6.2
104	Support for hyperlinking	SMIL Animation	3.6.5
105	Timing specifiers syntax	SMIL Animation	3.6.7
106	Animation events	SVG 1.1	16.11

**Table A.1 (continued)**

<b>ID</b>	<b>Title</b>	<b>Source</b>	<b>Section</b>
107	The timing model must handle the edge cases described in the SMIL Animation spec	SMIL Animation	3.6.8
108	The timing model should detect cyclic dependencies	SMIL Animation	3.6.8, Cyclic dependencies in the timegraph, Detecting Cycles
109	hasFeature should reflect supported interfaces	SMIL Animation	6.2

# Appendix B

## Test results

This appendix contains a selection of the test cases used in unit testing this project. Not all test cases are shown here due to space limitations. Instead the most important cases from the animation and timing model respectively have been selected for inclusion here.

### B.1 Animation function tests

The animation function tests cover the functionality in `nsSMILAnimationFunction`. This represents most of the complexity in the animation model.

#### B.1.1 TestSampleAt

This case tests the most simple sampling behaviour.

**Test parameters**

*from:* 5

*to:* 9

*duration:* 2,000ms



Sample time (ms)	Expected result	Notes
0	5.0	
500	6.0	
1999	8.5~9	

### B.1.2 TestParameterPriority

For different combinations of the `from`, `to`, `by`, and `values` attributes SMIL Animation defines which values take precedence. This case tests that behaviour.

#### Test parameters

*duration:* 5,000ms

Sample time (ms)	Expected result	Notes
5	0	No values set.
Set <i>from</i> to 5		
10	0	Only <i>from</i> is set.
Set <i>to</i> to 10		
1000	6.0	After setting <i>to</i> the animation is valid.
Set <i>by</i> to 10		
2000	7.0	<i>by</i> is overridden by <i>to</i> .
Unset <i>to</i>		
2000	9.0	Un-setting <i>to</i> lets <i>by</i> take effect.
Unset <i>by</i>		
3000	0	Animation is now in error.
Set <i>by</i> to 10. Unset <i>from</i> . Set base value to -5.0.		
2000	-1.0	By animation.
Set <i>to</i> to 20		
4000	15.0	To animation.
Set <i>values</i> to ' 40 ; 60; 50;100; -10; -20 '		

Sample time (ms)	Expected result	Notes
0	40.0	Values animation.
500	50.0	
1000	60.0	
3000	100.0	
<last value>	-20.0	
Unset <i>values</i>		
4500	17.5	Return to 'to animation'.

### B.1.3 TestValuesParsing

This case tests that bad values strings are correctly rejected.

Good specs	Bad specs	Notes
'10'	'10; '	
' 40 ; 60; 50;100; -10; -20 '	'10;'	
	'30;;40'	
	'30;73;40'	73 is defined to be a bad value in the test code
	';10'	
	'11;23 40;50'	

### B.1.4 TestAdditive

This case tests additive behaviour.

**Test parameters***duration:* 2000ms

---

Function A

---

*from:* 5*to:* 10*document position:* 0

---

Function B

---

*from:* 5*to:* 10*document position:* 1

<b>Sample time (ms)</b>	<b>Expected result</b>	<b>Notes</b>
Set additive to 'sum'		
1000	15.0	Simple additive behaviour.
Set additive to 'replace '		
1000	7.5	Simple non-additive behaviour.
Set additive to ' sum '		
1000	15.0	Test stripping of whitespace.
Set additive to 'sum a '		
1000	15.0	Check additive behaviour remains unchanged after error.

**B.1.5 TestKeyTimes**

This case tests the behaviour of the `keyTimes` attribute.

**Test parameters***duration:* 10,000ms*values:* 0; 50; 100

Sample time (ms)	Expected result	Notes
Set keyTimes to '0; .8; 1'		
4000	25.0	Test the simple case.
8000	50.0	
9000	75.0	
Set keyTimes to ' 0;.8;1 '		
2000	12.5	Test whitespace handling.
Set keyTimes to ';0; .8; 1'		
2000	0.0	Test leading semi-colon causes an error.
Set keyTimes to '; .8; 1'		
2000	0.0	Test leading semi-colon again.
Set keyTimes to ''		
2000	0.0	Test the empty string.
Set keyTimes to ''		
2000	0.0	Test the empty string again.
Unset keyTimes		
4000	40.0	
Set keyTimes to '0; .8'		
4000	0.0	Not enough keyTimes.
Set keyTimes to '0; .8; .9; 1'		
4000	0.0	Too many keyTimes.
Set keyTimes to '0; 1; .8'		
4000	0.0	keyTimes are not in ascending order.
Set keyTimes to '0; .8; 1.1'		
4000	0.0	keyTimes must be between 0 and 1 inclusive.
Set keyTimes to '0; -0.8; 1'		
4000	0.0	keyTimes must be between 0 and 1 inclusive.
Set keyTimes to '0; .8; 1'		
4000	25.0	Test recovering from an error.
Set <i>values</i> to '0; 50'. Set keyTimes to '0; 1'.		

Sample time (ms)	Expected result	Notes
6000	30.0	Test two values and two keyTimes.
Set <i>values</i> to '50'. Set keyTimes to ' 0'.		
7000	50.0	Test a single value and a single keyTime.
Set keyTimes to '1 '		
8000	50.0	Test a single value and a single keyTime again.
Set <i>calcMode</i> to 'paced'. Set <i>values</i> to '0; 50; 100'. Set keyTimes to '-1; 0; 1; 34; abc '.		
—	—	Test no error is reported.
Set <i>calcMode</i> to 'linear'.		
6000	0.0	Bad keyTime spec is not valid for <i>calcMode</i> == 'linear'.
Unset <i>calcMode</i> . Unset <i>values</i> . Set <i>from</i> to 10. Set <i>to</i> to 20. Set keyTimes to '0.0; 1.0'.		
3500	13.5	Test from-to animation.
Set keyTimes to '0.0; 0.7'		
3500	15.0	Test bad example from SMIL Spec.
7000	20.0	
9000	20.0	
Set keyTimes to '0.5; 1.0'		
1500	10.0	Test bad spec is allowed.
4999	10.0	
5000	10.0	
6000	12.0	
Set keyTimes to '0.2; 0.4'		
1999	10.0	Test a very bad spec is allowed.
2000	10.0	
3000	15.0	
4000	20.0	
9999	20.0	
Set keyTimes to '0.4; 0.4'		

Sample time (ms)	Expected result	Notes
3999	10.0	Test another bad spec is allowed.
4000	20.0	
4001	20.0	
Unset <i>from</i> . Set <i>keyTimes</i> to '0.0; 1.0'.		
3500	7.0	Test 'to animation'.

### B.1.6 TestKeySplines

This case tests the behaviour of the `keySplines` attribute.

#### Test parameters

*duration:* 4,000ms

Sample time (ms)	Expected result	Notes
Set <i>values</i> to '10; 20'. Set <i>keyTimes</i> to '0; 1'. Set <i>calcMode</i> to 'spline'. Set <i>keySplines</i> to '0 0 1 1'		
0	10.0	SMIL Animation example (a).
1000	12.5	
2000	15.0	
3000	17.5	
<last value>	20.0	
Set <i>keySplines</i> to '.5 0 .5 1'		
0	10.0	SMIL Animation example (b).
1000	11.0	
2000	15.0	
3000	19.0	
Set <i>keySplines</i> to '0 .75 .25 1'		
0	10.0	SMIL Animation example (c).
1000	18.1	

Sample time (ms)	Expected result	Notes
2000	19.3	
3000	19.8	
Set keySplines to '1 0 .25 .25'		
0	10.0	SMIL Animation example (d).
1000	10.1	
2000	10.6	
3000	16.9	
Set <i>values</i> to '10; 20; 30'. Set <i>keyTimes</i> to '0; 0.25; 1'. Set keySplines to '0 0 1 1; .5 0 .5 1'.		
500	14.99~15.01	Test multiple keySpline intervals.
999	19.95~20.05	
1000	19.99~20.01	
1001	19.99~20.01	
2500	24.99~25.01	
3250	28.8~29.2	
Unset <i>keyTimes</i>		
1000	14.99~15.01	Test the same with keyTimes.
1999	19.95~20.05	
2000	19.99~20.01	
2001	19.99~20.01	
3000	24.99~25.01	
3500	28.8~29.2	
Set keySplines to '0 .75 0.25 1 ; 1, 0 ,.25 .25 \t'		
3500	26.8~27.1	
Set <i>values</i> to 5. Set keySplines to '0 0 1 1'.		
0	5.0	Test a single value.
1500	5.0	
Unset <i>values</i> . Set <i>from</i> to 10. Set <i>to</i> to 20. Set keySplines to '.5 0 .5 1'.		
0	10.0	Test from-to animation.

Sample time (ms)	Expected result	Notes
1000	10.9~11.1	
Set keySplines to ‘.5 0 .5 1; 0 0 1 1’		
1500	0.0	Test giving too many splines produces no result.
Unset <i>from</i> . Set keySplines to ‘.5 0 .5 1’.		
0	0	Test ‘to animation’.
1000	1.8~2.2	

In addition to these test cases the following erroneous specifications have been tested to ensure the parser rejects them.

Specification	Notes
‘0,1.1,0,0’	Range checking.
‘0,0,0,-0.1’	Range checking.
‘ 0 0 , 0 0 , ’	Stray comma.
‘1-1 0 0’	Path-style separators.
‘0 0 0’	Wrong number of values.
‘0 0 0 0 0’	Wrong number of values.
‘0 0 0 0 0 0 0 0’	Wrong number of values.
‘0 0 0 0; 0 0 0’	Wrong number of values.
‘0 0 0; 0 0 0 0’	Wrong number of values.
‘0 0 0 0;’	Trailing semi-colon.
‘0 0 0; 0’	Misplaced semi-colon.
‘;0 0 0 0’	Misplaced semi-colon.

### B.1.7 TestAccumulate

This case tests the behaviour of repeating animations that accumulate their result.



**Test parameters***duration:* 10,000ms*values:* 0; 100; 120

Sample time (ms)	Repetition	Expected result	Notes
4000	0	80.0	Test default behaviour.
4000	1	80.0	
Set accumulate to 'sum'			
4000	0	80.0	Test sum behaviour.
4000	1	200.0	
<last value>	1	240.0	
Set accumulate to 'none'			
4000	1	80.0	
Unset accumulate			
5000	1	100.0	
Unset to			
7500	0	30.0	Test 'to animation'.
7500	1	30.0	

**B.1.8 TestByAnimation**

This case is just a further test to ensure 'by animation' operates as expected.

**Test parameters***duration:* 2,000ms*base value:* 50.0*by:* -50

Sample time (ms)	Expected result	Notes
0	50.0	
1000	25.0	

## B.2 Timed element tests

The timed element tests cover the functionality in `nsSMILTimedElement`. This represents most of the complexity in the timing model.

### B.2.1 TestOffsetStartup

This case tests the most simple offset start time.

#### Test parameters

*begin:* 3s

Document time (ms)	Expected sample time (ms)	Notes
0	0	
4000	1000	

### B.2.2 TestMultipleBegins

This cases tests the resolution of intervals when multiple offset values are provided in a begin specification.

#### Test parameters

*begin:* 2s; 6s

*duration:* 1s

Document time (ms)	Expected sample time (ms)	Notes
0	0	
2999	999	
3000	—	
6100	100	

### B.2.3 TestNegativeTimes

This cases tests that intervals that begin and end before the document begin are discarded.

#### Test parameters

*begin:* -3s; 1s ; 4s

*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
0	—	
1000	0	
1001	1	
2000	1000	
3000	—	
4007	7	

### B.2.4 TestSorting

This case tests the sorting of multiple offset times.

#### Test parameters

*begin:* -3s; 110s; 1s; 4s; -5s; -10s

*end:* 111s; -5s; -15s; 6s; -5s; 1.2s

*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
0	—	
999	—	
1005	5	
1200	—	

Document time (ms)	Expected sample time (ms)	Notes
5999	1999	
6000	—	
110100	100	
111000	—	

### B.2.5 TestZeroDurationIntervals

This case tests the handling of intervals of zero-duration. In the first case the zero length interval should play, followed by a second interval starting at the same point. No interval should play starting at 4s however, as there is no valid end for it. After adding a valid end ('indefinite') the interval at 4s should play as well.

#### Test parameters

*begin:* 1s ;4s

*end:* 1s; 2s

*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
0	—	
1000	0	
1001	1	
1999	999	
2000	—	
3000	—	
4007	—	

---

Set *end* to '1.1s; indefinite'

---

0	—	
1000	0	

Document time (ms)	Expected sample time (ms)	Notes
1001	1	
3000	—	
4007	7	

### B.2.6 TestMoreZeroDurationIntervals

These tests are designed to cover the cases where the pseudocode in the SMIL Animation specification section 3.6.8 reads:

```
If tempEnd == tempBegin && tempEnd has already been used in
  an interval calculated in this method call
{
  set tempEnd to the next value in the end list that is > tempEnd
}
```

So, whether or not a zero length interval is included will depend on if it has already been used in a previous calculation.

#### Test parameters

*begin:* -2s; -0.5s

*end:* -0.5s; 1s

*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
5	505	Previous begin, skip zero-length interval.
1001	—	

**Test parameters***begin:* -0.5s*end:* -0.5s; 1s*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
5	—	No previous begin, include zero-length interval.

**Test parameters***begin:* -2s; -0.5s*end:* -0.5s; -0.5s; 1s*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
5	—	Two end points that could make a zero-length interval, so skip the first and include the second.

**Test parameters***begin:* 0.5s*end:* 0.5s*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
500	—	Just a miscellaneous test to ensure the end-point exclusive timing is working.

### B.2.7 TestEndSpecs

This case tests that end specifications are used in the calculation of the current interval.

**Test parameters**

*begin:* 1s; 4s

*end:* 1.05s; 4.5s

*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
0	—	
1001	1	
1999	—	
3500	—	
4007	7	

### B.2.8 TestBlankBegin

Tests that when no begin spec is provided the default of zero is used.

**Test parameters**

*end:* 1s

*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
0	0	
500	500	
1000	—	

### B.2.9 TestIndefiniteBegin

Tests the behaviour when an indefinite begin specification is used.

**Test parameters**

*begin:* indefinite

*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
0	—	
500	—	
3000	—	

### B.2.10 TestBadInput

Tests the behaviour when an erroneous begin specification is provided.

**Test parameters**

*begin:* ‘some random string’

*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
0	—	
500	—	
3000	—	

### B.2.11 TestUnsetting

This cases tests that when attributes are un-set the state of the animation is correctly updated.



**Test parameters***begin:* 2s*duration:* 2s

Document time (ms)	Expected sample time (ms)	Notes
500	—	
Unset <i>begin</i>		
600	600	
Set <i>end</i> to 1s		
1200	—	
Unset <i>end</i>		
1500	—	
Set <i>begin</i> to 4s. Unset <i>dur</i>		
10000	6000	

**B.2.12 TestRepeatSpecs**

This cases tests the parsing of the `repeatCount` and `repeatDur` specifications.

Following are valid `repeatCount` specifications:

Specification	Notes
'1'	
'1.0'	
'0.001'	
'99999999.99999'	
'indefinite'	
' indefinite '	Whitespace handling.

Following are invalid `repeatCount` specifications:

Specification	Notes
'-1.0'	Negative repeat count.

<b>Specification</b>	<b>Notes</b>
'0'	Zero repeat count.
'0.0'	Zero repeat count.
'0.0000000'	Zero repeat count.
'0.0001'	Zero repeat count (as we only store to three decimal places).
'media'	Not allowed for a repeat count.
'junk'	Random junk.
'indefinite;'	Stray semi-colon.
'indefinite 3 '	Trailing junk.
'	Empty string.

Following are valid `repeatDur` specifications:

<b>Specification</b>	<b>Notes</b>
'3s'	
'0s'	
'indefinite '	

Following are invalid `repeatDur` specifications:

<b>Specification</b>	<b>Notes</b>
'-3s'	Negative duration.
'+3s'	Signed notation is not permitted.
'media'	Not allowed for a repeat duration.
'	Empty string.

### **B.2.13 TestRepeating**

This case tests basic repeating behaviour.

#### **Test parameters**

*duration:* 2s

*repeat count:* 2

Document time (ms)	Expected sample time (ms)	Expected repeat count	Notes
0	0	0	
500	500	0	
2000	0	1	
3000	1000	1	
5000	—	—	
Set repeat count to 2.2			
4399	399	2	
4400	—	—	
Unset repeat count. Set repeat duration to 4.5s.			
4499	499	2	
4500	—	—	

### B.2.14 TestFillMode

This case tests the behaviour of fill modes (i.e. frozen behaviour).

#### Test parameters

*duration:* 2s

Document time (ms)	Expected sample time (ms)	Expected repeat count	Notes
0	0	0	No fill mode. Client should be active.
1999	1999	0	
2000	—	—	Client should now be inactive.
2001	—	—	
Set fill mode to 'freeze'.			
0	0	0	Fill with no repetition. Client should be active.
1999	1999	0	
2000	<last value>	0	Client should now be inactive.

Document time (ms)	Expected sample time (ms)	Expected repeat count	Notes
2001	—	—	
Set repeat duration to 2.5s.			
0	0	0	Fill with non-integral repetition. Client should be active.
1999	1999	0	
2000	0	1	
2001	1	1	
2499	499	1	
2500	500	1	Client should now be inactive.
2501	—	—	
Set repeat duration to 4s.			
0	0	0	Fill with integral repetition. Client should be active.
1999	1999	0	
2000	0	1	
2001	1	1	
3999	1999	1	
4000	<last value>	1	Client should now be inactive.
4001	—	—	
Unset repeat duration. Set <i>begin</i> to -0.5s.			
0	500	0	Test negative start. Client should be active.
1499	1999	0	
1500	<last value>	0	Client should now be inactive.
1501	—	—	
3999	1999	1	

**B.2.15 TestRestartMode**

This case tests the functioning of the restart mode.

**Test parameters**

*begin:* 2s; 4s

*duration:* 3s

Document time (ms)	Expected sample time (ms)	Notes
0	—	Test the simple case.
3000	1000	
3999	1999	
4000	0	
6999	2999	
7000	—	
Set duration to 3s.		
0	—	Skip the early end.
3000	1000	
3500	1500	
4500	500	
Set restart to 'always'		
0	—	
6999	2999	
Set restart to 'whenNotActive'		
0	—	
3000	1000	
3999	1999	
4000	2000	
6999	—	
Set restart to 'never'		
0	—	

Document time (ms)	Expected sample time (ms)	Notes
3000	1000	
3999	1999	
4000	2000	
6999	—	

### B.2.16 TestMin

This case tests the behaviour of the `min` attribute.

#### Test parameters

*duration:* 3s

*end:* 2s

Document time (ms)	Expected sample time (ms)	Notes
1000	1000	No min.
2500	—	
Set min to 3s		
1000	1000	
2500	2500	
3500	—	
Set min to 'media'		
1000	1000	Media should be ignored.
2500	—	
Set min to 3s and then to 0s		
1000	1000	Setting min to 0s should have the effect of resetting it.
2500	—	
Set min to 3s then to -1s		

Document time (ms)	Expected sample time (ms)	Notes
1000	1000	min < 0 generates an error but is then ignored.
2500	—	
Set min to indefinite		
1000	1000	Indefinite min generates an error but is then ignored.
2500	—	
Set min to 3s		
1000	1000	Un-setting min before reaching the min condition.
Unset min		
2500	—	Un-setting min before reaching the min condition causes the interval to be constrained by the end specification.
Set min to 3s		
1000	1000	
2500	2500	
Unset min		
2600	—	Un-setting min after reaching the min condition will cause it to be constrained by the end specification.
Set end to 4s. Set repeat count to 2. Set min to 5s.		
1000	1000	Test with repeat.
4000	1000	The repeat count should be 1.
4999	1999	The repeat count should be 1.
5000	—	
Set end to 2s. Set min to 4s. Un-set repeat count.		
1000	1000	Test simple duration < min.
2500	2500	
3500	—	
Set end to 4s. Set repeat duration to 6s. Set min to 8s.		

Document time (ms)	Expected sample time (ms)	Notes
1000	1000	Test repeat duration < min.
5000	2000	The repeat count should be 1.
7000	—	
Set <i>begin</i> to '0s; 7s'. Set <i>end</i> to '4s; indefinite'.		
1000	1000	Test restart can still occur.
5000	2000	The repeat count should be 1.
7000	0	The repeat count should be 0.
8000	1000	
Unset <i>begin</i> . Unset <i>end</i> . Set duration to indefinite. Set min to 3s.		
1000	1000	Test indefinite interval.
2500	2500	

### B.2.17 TestMax

This case tests the behaviour of the `max` attribute.

#### Test parameters

*duration:* 3s

Document time (ms)	Expected sample time (ms)	Notes
1000	1000	No max.
2500	2500	
Set max to 2s		
1000	1000	Test a simple case.
2500	—	
Set max to 2s then to 'media'		
1000	1000	'media' should be ignored.
2500	2500	



Document time (ms)	Expected sample time (ms)	Notes
Set max to 2s then to 0s		
1000	1000	0 generates an error but is then ignored.
2500	2500	
Set max to 2s then to -1s		
1000	1000	max < 0 generates an error but is then ignored.
2500	2500	
Set max to 2s then to indefinite		
1000	1000	Indefinite is allowed.
2500	2500	
Set max to 2s		
1000	1000	
Unset max		
2500	2500	Un-setting max before reaching the condition allows the interval to continue.
Set max to 2s		
1000	1000	
2500	—	
Unset max		
2600	—	Un-setting max after reaching the condition cannot extend the interval.
Set max to 4s. Set repeat count to 1.5.		
1000	1000	Test constraining the simple duration.
3000	0	The repeat count should now be 1.
3999	999	
4000	—	
Set <i>begin</i> to 2s		
1000	0	Test begin offset is applied correctly.
3000	1000	The repeat count should now be 1.
5000	0	

Document time (ms)	Expected sample time (ms)	Notes
5999	999	
6000	—	
Unset repeat count. Unset <i>begin</i> . Set the duration to indefinite. Set max to 2s.		
1000	1000	Test with indefinite interval.
2500	—	
Unset max		
1000	1000	
2500	2500	
Set max to 2s		
3000	—	Test constraining an indefinite interval after the fact.

### B.2.18 TestMinAndMax

This case tests the behaviour of the `min` and `max` attributes in combination.

#### Test parameters

*duration:* 3s

Document time (ms)	Expected sample time (ms)	Notes
Set <i>end</i> to 2s. Set min to 3s. Set max to 2s.		
1000	1000	min > max.
2100	—	
Unset <i>end</i> . Set min to 2.5s. Set max to 2s.		
1000	1000	min > max again.
2600	2600	
Set min to 2.5s. Set max to 2.5s.		
1000	1000	min == max.
2499	2499	

<b>Document time (ms)</b>	<b>Expected sample time (ms)</b>	<b>Notes</b>
2500	—	
<hr/> Set <i>end</i> to 2s. Set min to 2.5s. Set max to 2.5s. <hr/>		
1000	1000	min == max again.
2499	2499	
2500	—	