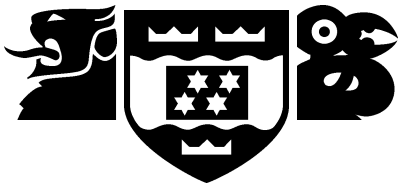
VICTORIA UNIVERSITY OF WELLINGTON
Te Whare Wananga o te Upoko o te Ika a Maui



Computer Science

PO Box 600
Wellington
New Zealand

Tel: +64 4 463 5341
Fax: +64 4 463 5045
Internet: office@mcs.vuw.ac.nz

Evaluating Scalable Vector Graphics for Software Visualisation

Matthew Duignan

February 19, 2003

Submitted to the Victoria University of Wellington in partial fulfilment of the requirements for the degree of Master of Science in Computer Science.

Abstract

Software visualisation employs various representations of software to help programmers better understand program code. However, there are many technologies that can be used to deliver software visualisations. These different software visualisation media have varying capabilities, and determining which medium is best suited for a particular software visualisation application can be a complex task. To this end, this thesis presents a principled model for evaluating software visualisation media. This model is then applied in the evaluation of the new "Scalable Vector Graphics" (SVG) standard, to determine if it is suited for use in a developing web-based software visualisation architecture. While the evaluation finds that SVG can realise a broad range of software visualisations, it is clear that it falls short in making the development of software visualisations as easy as it could. This thesis presents a way forward for creating complex software visualisations with SVG through the development of a domain-specific SVG library. The foundation for this library is illustrated and discussed.

Acknowledgments

Many thanks to Robert Biddle for his patience and advice as well as Ewan Tempero, Mike McGavin, Stuart Marshall, Kirk Jackson, and the rest of the elvis group for the many enjoyable hours of discussion. A big thank you to Paul and Jens for letting me live in their office and completely hog the best computer — and to Tim, sorry you couldn't play any games! Thanks to Sharni for being understanding and letting me get away with being distracted and grumpy, as well as reading this whole thesis. A big thank you to Joan Skinner, Paul Duignan, Craig Anslow and Peter Glensor for helping with proof-reading. Thanks to my friends for your support. Lastly, thank you to all of my family who have been so supportive, and humoured me when I wanted to show you what I was doing!

Contents

Chapter 1

Introduction

Mackinlay et al.[7] make the distinction between using vision to communicate, and using vision to think. We can make a similar distintion in software visualisation. Software visualisation can be used to document software for learning (using vision to communicate), and to aid in the design process (using vision to think). Using software visualisation to document software aids both the maintenance and reuse of program code by making it easier to understand. Using software visualisation as part of the design process helps designers to comprehend the complexity of the systems they are building, and helps them to see the patterns and relationships as they emerge. To utilise visualisation for these ends requires software technology capable of high-quality graphics, interaction and dynamic display. What is more, visualisations should be able to be generated automatically from the underlying software, demanding that the visualisation technology integrates with other technologies involved in extracting software information and mapping it to graphics.

The Visualisation Architecture for REuse (VARE) being developed by our research group at Victoria University of Wellington seeks to specify a system for the support of code reuse through using visualisation as documentation[4]. This architecture is designed to support web-based code repositories by providing a means to "test drive" code and view visualisations over the web. VARE is in need of a software technology to deploy these visualisations over the web.

"Scalable Vector Graphics" or SVG[61], is a new standard for describing graphics in the eXtensible Markup Language (XML)[60]. On first examination, SVG seems to fill many of the requirements for VARE's needs. It can be deployed over the web, supports interaction and animation, and it can display high-quality graphics. However, SVG needs to be examined more thoroughly.

This thesis develops a model for evaluating software visualisation "media". This model itemises a set of capabilities which are desirable in a software visualisation display technology. The model is then used as a basis for evaluating SVG for software visualisation, with the end goal of discovering how appropriate SVG is for use in VARE.

This thesis also has the goal of testing various elements of the VARE architecture. This

includes testing VARE's method for the programmatic generation of software visualisations, building from the work that has already been done by our research group[4][38].

The evaluation methodology for SVG undertaken in this thesis is as follows:

- Step One: Develop a model of required capabilities for a software visualisation display technology.
- Step Two: Examining SVG's general capabilities.
- Step Three: Construct software visualisation examples in SVG to test software visualisation specific capabilities.
- Step Four: Evaluate SVG's capabilities against the capabilities identified in the evaluation model.

The organisation of this thesis is as follows:

- Chapter 2: Covers the background of information and software visualisation, and gives a description of the VARE architecture. This chapter also explains the rationale for this thesis.
- Chapter 3: Introduces, argues for, and explains the evaluation model that I have developed. (Step one of the methodology).
- Chapter 4: Covers the steps that were taken in exploring the details of SVG. This chapter includes both learning SVG as well as creating real software visualisation examples. (Steps two and three of the methodology).
- Chapter 5: Contains the comparison of SVG's capabilities as identified in chapter 4, with the capabilities identified in the model developed in chapter 3. (Step four of the methodology).
- Chapter 6: Examines how SVG could be augmented without requiring changes to its existing standard. This examination takes into account the weaknesses that were identified in chapter 5. This chapter then describes the development of one such augmentation, an SVG software visualisation library, and discusses the results.
- Chapter 7: Draws conclusions from the findings of chapters 5 and 6 for both SVG and VARE.

Chapter 2

Background

One of the biggest strains for software developers today is trying to understand existing program code. Programmers must face this task in several situations. The most obvious of these are in maintaining code and when reusing code. For the task of maintaining code, programmers need to rapidly come to understand the general architecture of a system, while gaining thorough knowledge of the interactions and behaviour of those specific components to which changes need to be made. In the case of code reuse, programmers want to gain access to code which solves a specific problem or performs a certain task. To accomplish this they need to locate the code, and discover exactly what it does. Often they will also be required to investigate parts of that code in detail, to discover if it must be modified to serve the demands of their application. In those cases where the code was written by the programmer who is doing the maintenance or reuse, they may come back to their code with only distant memories of how it works. Worse still, programmers will often have to deal with code written by someone else. In these situations, how do programmers learn how the code works? The common-sense answer is that they read the source code. The problem with this is that while the original author(s) had the benefit of a developed multi-leveled mental model of the software while they worked, the newcomer has to construct a model as they go. The overview that the programmer needs before delving deeper will be difficult to construct through studying source code alone. Even a cursory examination of the scores of source code files that might make up a single component of a system can be laborious and painful. Such an examination is also likely to fail to give the reader a good understanding of overall relationships and global structure. Is constructing this sort of higher-level mental model best served through textual code and textual documentation, or can we do better?

2.1 Object-oriented programming

Object-oriented programming aims to alleviate this problem by helping programmers capture the mental models of programs that they hold in their minds, and represent them in a concrete form in the semantics and syntax of their programming languages. In the object-oriented

paradigm, all functionality is captured in "Objects" which act as active entities in a program's execution. All interaction between Objects is mediated by message passing. The predefined messages that an Object will recognise and respond to are called "methods". A "method-call" on an object may be a request for that object to do some specific work. At this point, this object can then do work itself or delegate its responsibility by calling methods on other objects. However, while enabling programmers to capture this structure in the syntax of their programming language helps with the problems described above, this is only one part of a more comprehensive solution. Once programmers are thinking and coding with objects, why can they not also *look* at them to *see* how they work as we can with objects in the real world?

2.2 Information visualisation

The development of information visualisation as an active field of research, and the success of the resulting visual tools, leads us to believe that the graphical medium can unlock the powerful potential of the human visual system. Using visualisation to communicate or store information is not a new practice. The mapping of the earth and the heavens on paper is an ancient example of visualisation. However, visualising more abstract information (i.e. other than spatial representations) was not begun in earnest until the work of Charles de Fourcroy's "Poleometric Table" in 1782, and the publication of "The Commercial and Political Atlas" by William Playfair in 1786 .

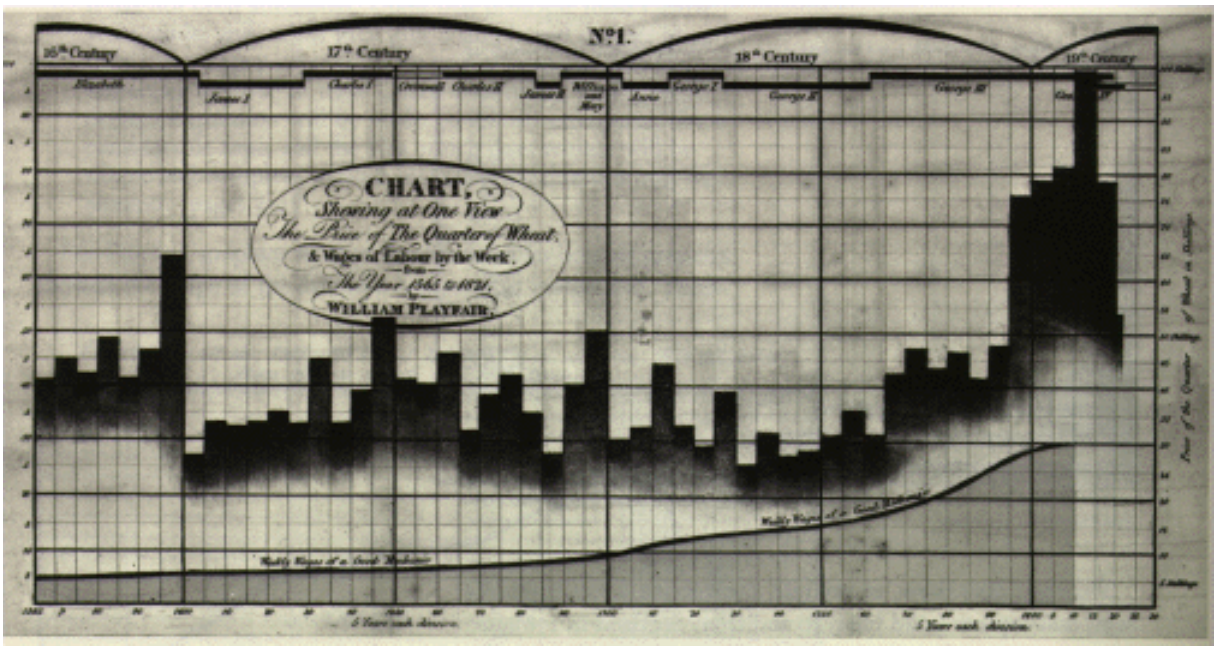


Figure 2.1: An example of early information visualisation from William Playfair's "The Commercial and Political Atlas".

Information visualisation is used for both communicating information, and as an aid to thinking about and understanding information. Mackinlay et al.[7](p.1) make the distinction be-

tween using vision to think and using vision to communicate. Visualisation is a powerful communication device due to the high bandwidth of our visual system[69](p.xviii). Visualisation helps us think because it allows us to extend our memories, group related information together visually, and find patterns in complex data[30].

The fields of scientific visualisation and information visualisation have developed greatly as our capability to generate high quality graphics on commodity computer hardware has increased. Being able to rapidly generate and interact with data in a visual form has created new opportunities in extending our analytical capabilities. When we bring this to bear on the problem that programmers face with understanding code, we are led to the field of software visualisation.

2.3 Software visualisation

The term software visualisation describes the application of information visualisation to the software domain. One early software visualisation tool was "SeeSoft". SeeSoft was designed to help programmers see trends in the physical layout of their code[19]. This corresponds to Mackinlay et al.'s first notion of using vision to think. Software visualisation can also help programmers to think by using visualisations in software design. Indeed, up till now, this has been the major success of software visualisation. Standards like ER (Entity Relationship) diagrams for visualising data structures, and UML (Unified Modelling language)[24] for visualising object-oriented systems have made significant inroads into real world software development. Such standards allow programmers to see the higher-level structure of programs visually. These visualisations aim to answer the following questions. What are the components of the system? How do they interact? How *can* they interact?

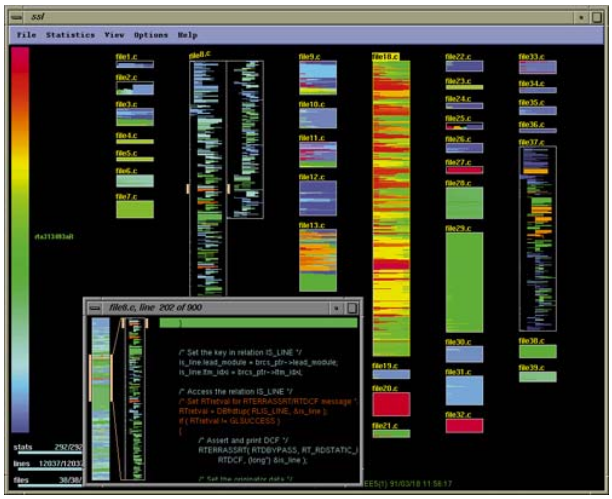


Figure 2.2: SeeSoft in action.

The other use of visualisation is what can be referred to as "visualisation as documentation" as used in the VARE architecture[4] described in the following section. This corresponds to

Mackinlay et al.'s second notion of using vision to communicate. But unfortunately, providing visualisations of existing software for visualisation as documentation has proved to be much more difficult than using it in software design. The reasons for this are twofold. Firstly, software is often developed without formal software visualisations. Perhaps the design is fleshed out on scraps of paper, or solely in the developers' heads. Secondly, even if the original software developers do create visualisations during the design of a system, software still tends to change significantly, both in the implementation and in the following maintenance periods. This makes the visualisations inconsistent with the actual reality of the final system.

An additional problem for visualisation as documentation is how to address the distributed nature of software reuse. With the increasing availability of free and reusable software components on the Internet, we are moving into an environment of increasing potential for software reuse. The development of standard libraries for generic and common reuse is one example of this, but there is still more potential for the specific and specialised code that is becoming available to play a useful role in reuse. But before we get this far, we have to ask; how can we find code that we can reuse, how do we check if it works in the way we require, and what resources are there to help us understand the code to the level of detail that we require? These are the questions that the VARE framework attempts to address.

2.4 VARE

The Visualisation Architecture for REuse (VARE) is a developing architecture designed by our research group to support web-based code reuse through visualisation[4]. VARE is an attempt to address the current deficiencies of stand-alone web-based code repositories. Web-based code repositories allow access to potentially vast collections of program code. Such code could vary from small classes to accomplish small tasks, through to entire frameworks for system development. While having access to such repositories presents amazing possibilities for the software development world, the speed at which programmers can understand the design and behaviour of a code component becomes crucial. VARE aims to allow users to examine code behaviour remotely, and provides the architecture to present them with high quality visualisations of the way the code behaves. Without such capabilities, web-based code repositories (for reuse) are in danger of becoming almost impossible to use.

2.4.1 Program Mapping Visualisation

The VARE architecture is based on the foundation of the Program Mapping Visualisation model (PMV) developed by Stasko[52], and Roman and Cox[13] and later formalised by Noble[45]. This conceptual model deals with the creation of visualisations programmatically from source code, or from a running program. The basic idea is to take a program, and extract information for visualisation purposes. This allows prospective users of code to create visualisations automatically without needing to know how a program works in advance.

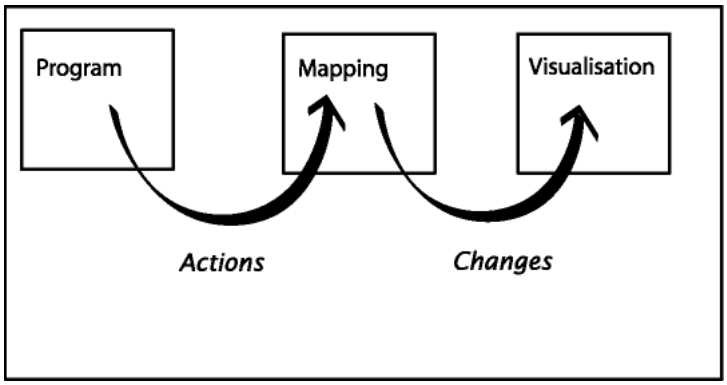


Figure 2.3: The Program Mapping Visualisation model of software visualisation creation.

Additionally, software developers can create visualisations as documentation for their own projects with a minimum of effort, and without extensive graphic knowledge.

The PMV model divides the process into the three core components suggested by its name: *program*, *mapping*, and *visualisation*. The *program* component is not the program being visualised itself, but is instead the component responsible for providing the rest of the system with information from inside the visualised program. The "...program component presents the target program to the visualisation system"[45](p.6). The *mapping* component takes information about the program to be visualised from the *program* component, transforms it, and passes it to the *visualisation* component. The nature of the transformation has varied between implementations, but discussing this is beyond the scope of this thesis. Finally, the *visualisation* component deals with the interaction with the user. This includes displaying the graphical elements, as well as collecting user input for interaction if needed.

2.4.2 PMV to VARE

VARE takes the PMV model and moves it into client/server territory for a web-based code repository environment. The first thing to note is the similarities between the PMV model and VARE. The *program*, *mapping* and *visualisation* components are all represented in VARE — albeit with different names. The PMV's *program* component is referred to as an "engine". This makes explicit the difference between the program being visualised and the component which makes the program behaviour available for visualisation. In the VARE architecture, PMV's *mapping* component is called the "transformer". Thirdly, the *visualisation* component is also included in VARE, this time retaining the same name. These elements are then made accessible via the Internet for clients to access.

2.4.3 Architecture

Figure 2.4 shows the current state of VARE's still developing architecture. There is an important distinction shown in this figure indicated by the horizontal line marked "Network".

This indicates the client/server divide, with the elements relevant to the client shown above and the server below.

Server

To provide the web-based code repository capabilities, the code to be visualised is stored on the server in a "Component Repository". This also implies, and therefore results in repositories for engines, transformers, and pre-constructed visualisations. This allows the architecture to support the heterogeneous and varied demands of multiple users, multiple programs, multiple programming languages, and multiple types of visualisation. When initiated, code components are loaded from the Component Repository for execution. A suitable engine is selected from the engine repository and attached to the Component. During the execution of the code components, the engine eavesdrops on their internal behaviour, and records this in a "test drive report". This can be streamed to a transformer, as well as recorded in a "Test Drive Report Repository". In a subtle difference from the PMV model, the VARE transformer takes the Test Drive Report and *creates* a "visualisation", which will later be rendered at the client side.

Client

The client side system is web-based, consistent with making the system easily accessible over the Internet. The client side presents interfaces to control the location and selection of code components to visualise; the setup and execution of test runs; the selection of transformers to create visualisations; and the interface to actually experience the visualisations. All of this complex activity is mediated through the metaphor of directing, filming and viewing a movie.

2.4.4 Communication

XML

Communication in VARE is dominated by XML[60] based technologies. The Extensible Markup Language (XML) has been described as "...the universal format for structured documents and data on the Web."[59] It provides a textual, human-readable, tree-based method of storing data. Data in XML is "marked up" with "tags". An "element" is delimited by a starting tag and a closing tag pair. Tags are a name contained between angle brackets. An opening tag might look like `<tagname>`. A closing tag has a forward slash before the name as in `</tagname>`. XML also allows empty elements. Such elements are shown as a tag with a forward slash after the name, such as `<tagname/>`. Each element can contain further sub elements which gives XML its tree structure. Actual data can be contained in the elements attributes or as data at the leaves of the tree. Attributes are contained after the elements tag name and contain an attribute name and value (in the form `name=value`). Leaf data in

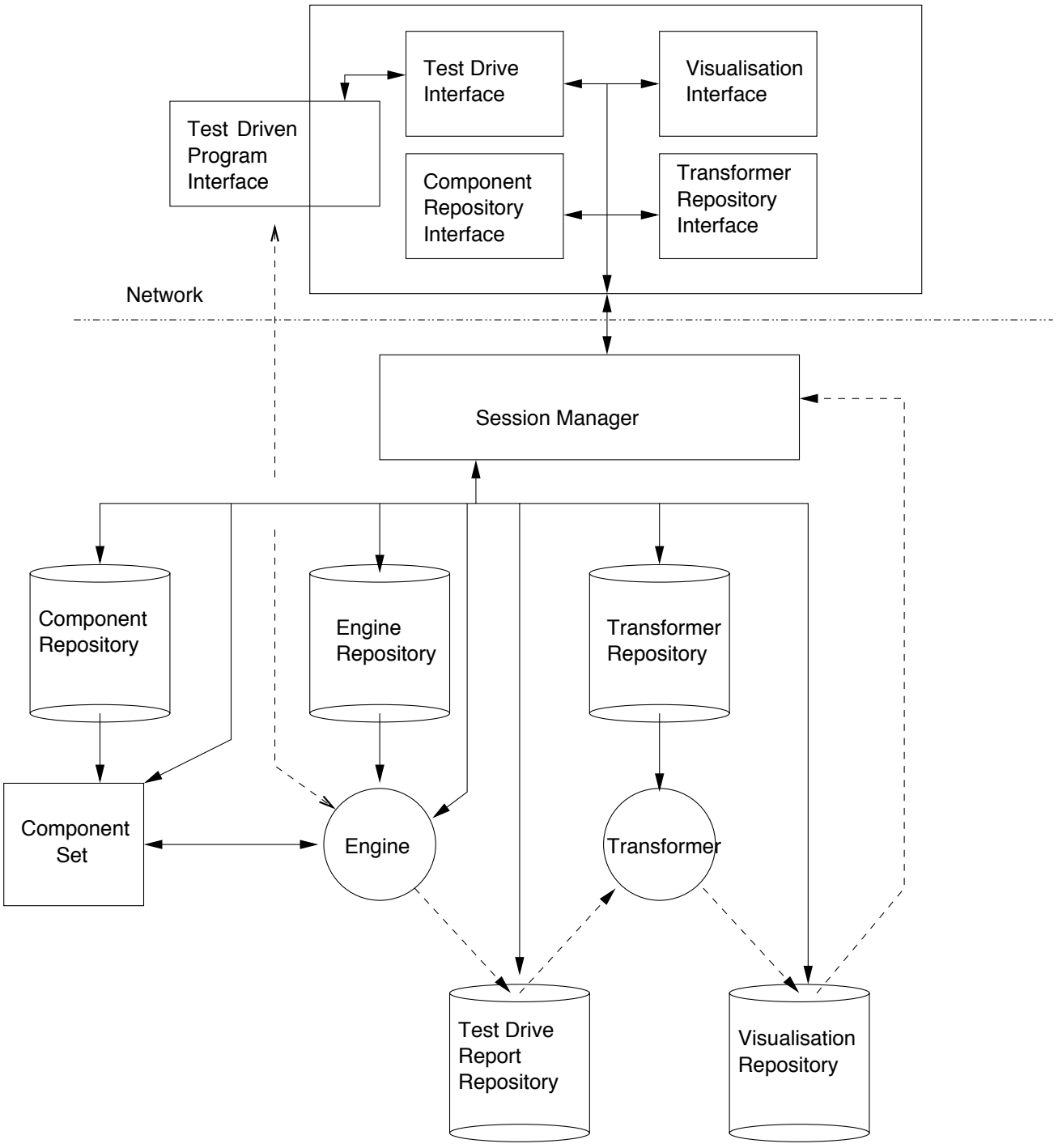


Figure 2.4: VARE architecture.

XML is simply the data between two tags which is not a sub-element. The approach of using these types of tags for data storage is called "markup" and XML as well as its relatives are therefore called "markup languages".

As an example of XML, a student could be represented by a `student` element containing `name`, `age`, `id` and `phonenumber` elements. The phonenumber element has a "status" attribute as to whether the phone number it represents is public or should be kept private. When written to a file this could look like figure 2.5.

```
<student>
  <name>James</name>
  <age>15</age>
  <id>1000100</id>
  <phonenumber status="secret">478-4483</phonenumber>
</student>
```

Figure 2.5: A simple example of XML showing data about a student.

The purpose of XML is to be *extensible*. This means that new XML based markup languages can be defined using the XML syntax. These definitions determine the elements that can be used, as well as when and how they can be used.

The benefit of XML lies in its status as a well-supported open standard. This ensures that the XML developer can leverage a huge set of standard tools and parsers that provide much of the common functionality required in data manipulation. For example, rather than work with a data file at the byte or character level, XML documents can be explored with tools that support higher-level abstractions. Two popular higher-level data access abstractions for XML are the "Document Object Model" (DOM)[58] and the "Simple API for XML" (SAX)[5].

- The Document Object Model

  The DOM specifies a set of interfaces which provide access to the underlying XML data. When implemented in a library, the DOM allows the programmer to explore XML data as a set of objects structured in the "tree" of the document. This fits nicely with the typical object-oriented programming approach. The underlying data is automatically transformed into the program objects without the need for the programmer to deal with file format details.

- The Simple API for XML

  Alternatively, the SAX allows the programmer to register interest in certain elements in an underlying XML document. As the XML document is parsed by the SAX implementing code it triggers events to be dealt with by a higher-level program.

  Both the DOM and SAX are implemented in a large number of public libraries for inclusion in new projects. Using these libraries removes a large burden in the development of the software that utilises them. This however is not XML's only strength.

XML provides a good foundation for building independent components in a larger system by allowing developers to define the XML languages which components must speak when communicating. The primary XML languages used in VARE are the Simple Object Access Protocol (SOAP)[64], and a new purpose-built language called the Process Abstraction Language (PAL)[38].

The Simple Object Access Protocol

The "Simple Object Access Protocol" (SOAP) is an XML-based language which is designed to enable communication between web services[64]. Using a standard such as this allows each service to be completely independent of the implementations of the others. This has obvious benefits for use in the VARE architecture which has multiple components which need to be implemented in a number of different languages. This is due to the fact that VARE will have to test drive programs in any language that needs to be visualised. While it has been decided that SOAP will be used in VARE for this purpose, further details are still being fleshed out by our research group. Further discussion of SOAP is outside the scope of this thesis.

The Process Abstraction Language

The Process Abstraction Language (PAL)[38] is being used as the storage form for the Test Drive Reports in VARE. PAL is designed to describe the execution of object-oriented code in a standard and easily accessible format. It can represent both static information, such as data and object types, as well as dynamic information, such as method-calls and new class instances. An example of a PAL document is shown in figure 2.6. The advantage of using the clearly defined syntax that PAL provides is that the engine components become independent from the transformer components. As long as an engine outputs its Test Drive Reports in PAL, it should be fully compatible with all existing PAL-reading transformers. This allows new engines and transformers to be added to VARE without having to check all components for compatibility.

However, while PAL was developed for the very purpose of being used in the VARE architecture, it was initially created for use in a C++ case study[38]. While every effort was made to make PAL as generic as possible it will certainly need to be extended as support for other programming language characteristics are added. One obvious extension to the current implementation of PAL would be to include exception events. More extensions of this nature seem likely, although the basic design of PAL allows it to be extended without breaking older components that do not recognise these new extensions.

However, it is important to note that PAL has been designed with object-oriented systems in mind. This does indicate that it might not be ideal for more atypical languages such as pure functional languages. PAL can however describe the executions of languages such as C which tend to contain a subset of the object-oriented language semantics, rather than being radically different.

```
<pal>
<execution>
<event eventid="_200">
 <processbegin>
  <argstring></argstring>
 </processbegin>
</event>
...
<type name="FoodItem" typeid="t3">
 <context contextname="sourcefile" contextvalue="tp7.cc"/>
 <classdata>
  <methods>
   <method access="public" methodid="f40" name="operator=">
   ...
    <argument argumentid="a39">
    ...
    </argument>
   </method>
  ...
  </methods>
 </classdata>
</type>
...
<event eventid="_222">
 <methodcall callermethodcallidref="dmc212" callerpositionref="dp215"
             classinstanceidref="dcl217" methodcallid="dmc220"
             methodidref="f27" threadnum="1"/>
</event>
...
<event eventid="_376">
 <methodreturn methodcallidref="dmc212"/>
</event>

<event eventid="_377">
 <processend/>
</event>

</execution>
</pal>
```

Figure 2.6: A reduced example of PAL output from a C++ engine. Inside the execution element are any number of type and event elements. In this example the FoodItem class is described in a type element, and a methodcall and methodreturn events occur as described by the event elements.

This leaves two questions to be answered to complete the communication framework of VARE. How will visualisations be transmitted to the client, and how will they be displayed?

2.4.5 Finding a medium for software visualisation over the web

VARE is in need of a technology to deliver the visualisations it creates. The selection of such a technology must involve several careful considerations. Firstly, the visualisations need to be created programmatically. VARE should be able to create visualisations automatically with a minimum of human intervention regarding graphical details. Secondly, the visualisations should be able to be stored in some form on the server so they can be easily retrieved. This is because VARE's architecture should allow the browsing of previously created visualisations without the need for a transformer. This is also required, as it must be possible for modifications or annotations to be made to visualisations after the transformation process, which can then be accessed by interested users. Thirdly, the visualisation needs to be transported to the client over the network. If this can be cleanly integrated into the other web-based technologies, this would be advantageous. Most importantly, the visualisation technology needs to have the core capabilities required to implement the visualisations.

2.4.6 The Unified Modelling Language

The Unified Modelling Language, known more commonly as UML, was briefly introduced in section 2.3. UML is a standard for modelling object-oriented systems in a diagrammatic form and was specified by the Object Management Group[24]. It has become an industry standard, being taught in universities and is required knowledge for many software engineering positions. As such, UML is one of the types of visualisations that VARE must be able to create. There are twelve diagram types in UML, some of which are purely for assisting designers, and others that can also be used for the "visualisation as documentation" purpose to which VARE is aimed. We will briefly discuss three of the latter type, with the aim of illustrating the kinds of visualisations that VARE must support as a minimum.

UML class diagrams

A UML class diagram describes the "types of objects in the system and the various kinds of static relationships that exist among them."[22](p.49). An example is shown in figure 2.7 from the Objects by Design website[14]. Class diagrams show two principal types of static relationships, associations between objects and hierarchies for the classification of types. Objects are shown as boxes with the object's name (or type), its attributes (data) and the operations that can be carried out on it (methods) listed as text inside. Relationships are displayed as various types of lines linking appropriate objects. Different line types include variations on arrows (with different types of arrow-head), dashed and full lines, as well as lines with special notations along their length.

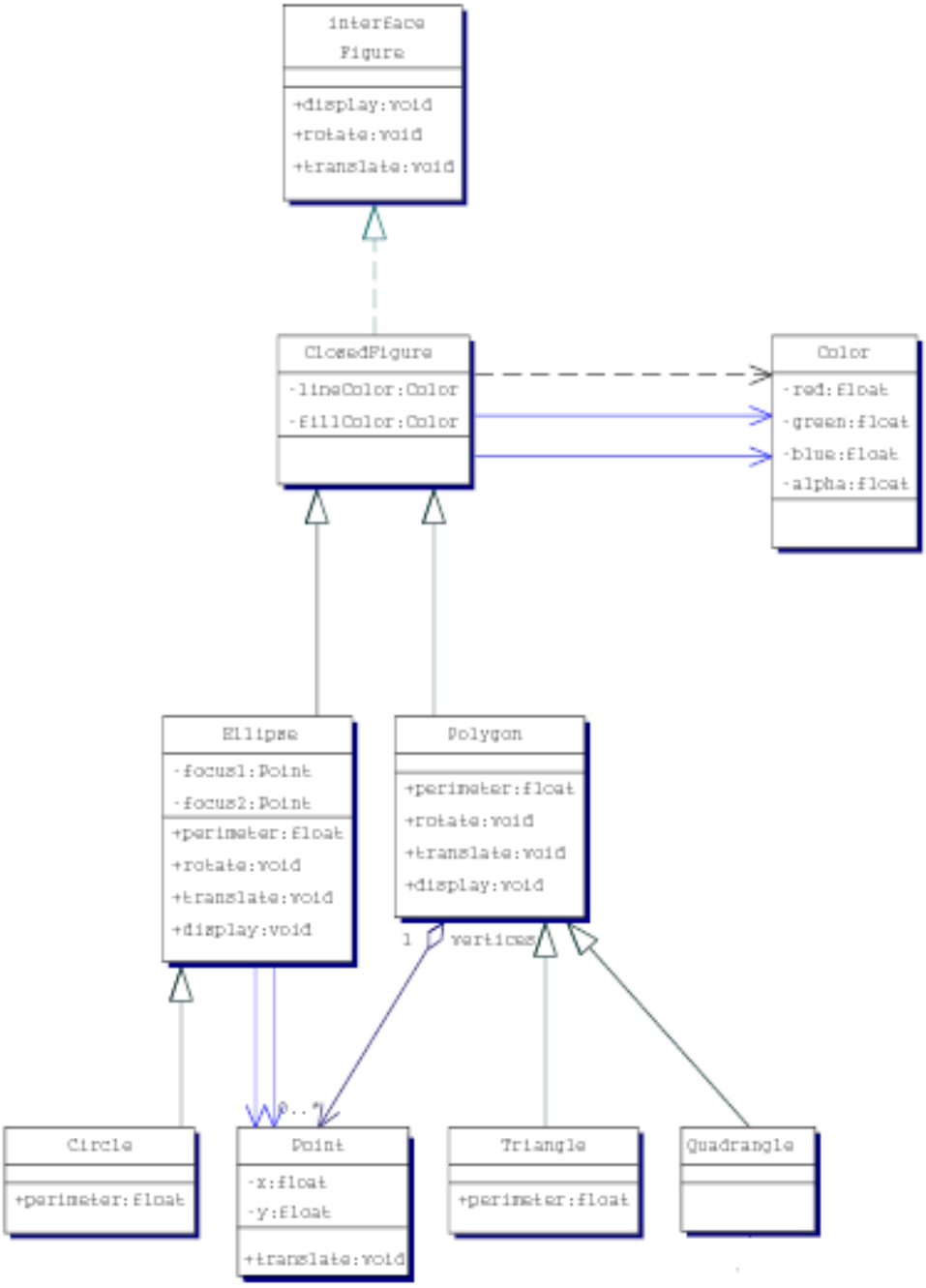


Figure 2.7: A UML class diagram. (Objects by Design)

UML sequence diagrams

A UML sequence diagram is a type of interaction diagram. Figure 2.8 from Rational Software Corporation[11] shows its basic form. Sequence diagrams show how the various objects in the system interact when a given scenario plays out. It consists of a series of the system's objects mapped across the x-axis, while time is mapped down the y-axis. Each object is shown as a box at the top of the diagram and has a "lifeline" which is shown as a vertical line from the box downwards through time. Interaction between objects (the subject of the diagram) is shown as arrows from one object's lifeline to another at the appropriate position in time. These interactions are method-calls and returns. Between a method-call and its return there is a method activation box, which is depicted as an elongated rectangle stretched along the lifeline for the duration of the wait. A method-call from one object may result in any number of consequent method-calls before a response is returned.

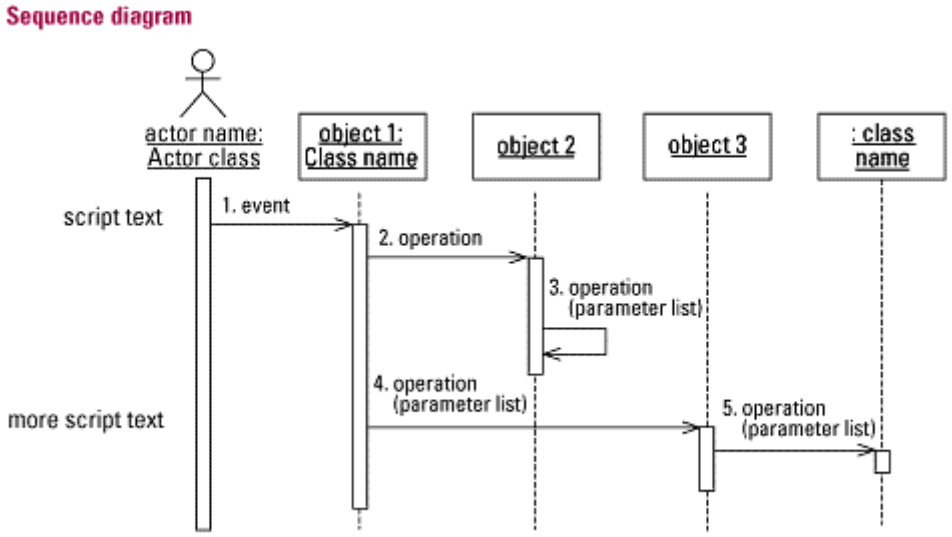


Figure 2.8: A summary of the UML sequence diagram. (Rational Software Corporation)

UML collaboration diagrams

UML collaboration diagrams are very similar to Sequence diagrams. However, they do not encode time spatially. Instead, objects are displayed in a convenient layout and method-calls between them are numbered to show ordering. Figure 2.9, again from Rational Software Corporation[11] shows the basic form of the collaboration diagram.

2.5 Scalable Vector Graphics

With the demands of VARE in mind, a new XML-based graphic language seems to be a potential fit. This language is called "Scalable Vector Graphics", or more commonly, "SVG".

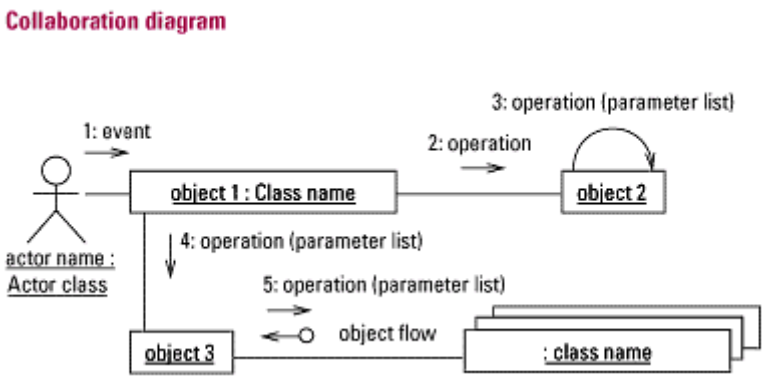


Figure 2.9: A summary of the UML collaboration diagram. (Rational Software Corporation)

It is a newly-created open standard, which reached version 1.1 on the 14th of January 2003[62]. In this section, the aim is to give the flavour of SVG and how it works, rather then to attempt to describe it in any real detail. The standard itself is a very large document and it is beyond the scope of this thesis to try and summarise it all here.

2.5.1 Bitmap versus vector graphics

As the name suggests, SVG is based on *vector* graphics rather than traditional bitmapped graphics. The difference is primarily in the content of the graphics file format. Bitmap graphics describe the properties of each individual point of colour that makes up the image. In a primitive implementation, a bitmap graphics file could describe a two dimensional grid, and specify the colour value for each square. When it is to be displayed on a screen, the program would place each square of colour in its correct place in an invisible grid on the screen.

In contrast to this, vector graphics describe the logical entities in a graphic. Instead of encoding each unit of colour (pixels), vector graphics files encode entities like curves, circles, and words. The structure of the image is maintained in its file format. Only when the vector graphics file is to be rendered on the display are the low level pixel values calculated. This has many benefits.

- Firstly, vector graphics can be more independent of their display medium. When a graphic is created and distributed, it could be displayed on a high or low resolution monitor, cell-phone, or even be converted to audio for a blind user. The graphic can be seamlessly scaled to an appropriate size for display. The important point is that each display device "understands" the graphic, and so can make intelligent decisions about how to display it in its particular situation.
- Secondly, vector graphics retain semantic information that is lost with bitmap graphics. An Internet search engine can potentially access vector graphics and index their textual content.

- Thirdly, because vector graphics do not resolve down to pixels until the last moment of display, they enable intelligent changes to be made long after they were originally created. For example, there is the potential to easily change the position of a square in the background of a picture, the colour of a border, the text of a label, or the shape of an arrow. This is the reason that most graphics editors' native file save formats are variations on the vector graphics principle.
- Fourthly, vector graphics can include bitmap graphics as elements inside themselves. Bitmap graphics cannot do the reverse.

While this only highlights some of the important differences between vector and bitmap graphics, it clearly shows some of the features that make vector graphics very promising from the perspective of VARE.

2.5.2 SVG in practice

The SVG XML language defines a set of element types which describe graphics. For example, there is a circle element that defines a circle to be drawn when the SVG is shown in a renderer. The code for this might look like figure 2.10. The attributes `cx` and `cy` specify the center coordinates while `r` and `style` define the radius and appearance of the circle respectively. The rendered result can be seen in figure 2.11.

```
<circle cx="100"
        cy="100"
        r="50"
        style="stroke:blue;
               fill:red;
               stroke-width:5"/>
```

Figure 2.10: SVG code to draw a circle.

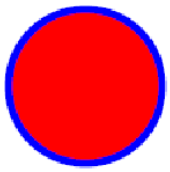


Figure 2.11: The circle from figure 2.10 displayed by an SVG browser plug-in.

In the XML way, SVG graphics are defined structurally, with lower level elements able to be combined into higher-level groups. An arrow can be defined by grouping a long black line and a triangular path for the head together in a `g` element, as shown in figures 2.12 and 2.13. This arrow group could be defined as a `symbol` which could then be `used`[1] in many places

[1] A `use` element can use a `symbol` element.

```
<g style="fill:black; stroke:black">
  <line x1="0"
        x2="70"
        y1="10"
        y2="10"/>
  <path d="M70 5 L70 15 L80 10 z"/>
</g>
```

Figure 2.12: SVG code which draws an arrow as a line and a path as shown in figure 2.13.

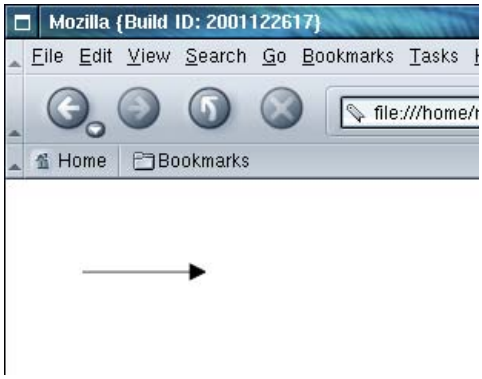


Figure 2.13: The arrow from figure 2.12 displayed in an SVG browser plug-in.

in the diagram. When the arrow is used, it can be stretched, rotated or moved arbitrarily. This is shown in figures 2.14 and 2.15.

```
<symbol id="Arrow" preserveAspectRatio="none" viewBox="0 0 80 20">
  <g style="fill:black; stroke:black">
    <line x1="0" x2="70" y1="10" y2="10"/>
    <path d="M70 5 L70 15 L80 10 z"/>
  </g>
</symbol>

<use height="20" width="80" x="50" y="0" xlink:href="#Arrow" />
<use height="20" width="80" x="50" y="40" xlink:href="#Arrow"
     transform="rotate(165 90 40)" />
<use height="20" width="20" x="50" y="60" xlink:href="#Arrow" />
<use height="100" width="80" x="50" y="60" xlink:href="#Arrow" />
```

Figure 2.14: SVG code that defines an arrow as a symbol, and then uses it four times with differing coordinates, sizes and transformations applied. The result can be seen in figure 2.15

The other highlights of SVG's functionality include animation, interactivity, and hyperlinking as well as a myriad of graphical display constructs such as filter effects and gradients. SVG's capacity is also hugely augmented by its inclusion of script elements. Script elements contain links to (or include in-line) executable code written in a scripting language. These scripts are interpreted at runtime, and can manipulate all aspects of the SVG currently being viewed by interacting with the DOM. The scripting language to be used is specified when in-

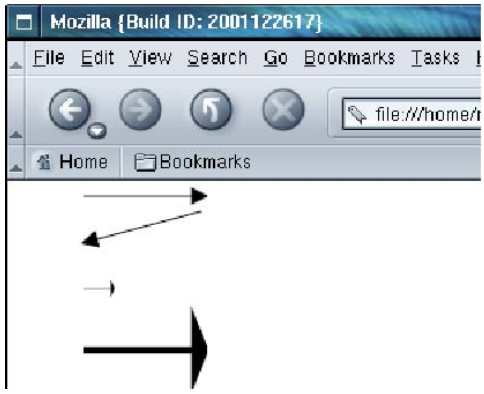


Figure 2.15: The SVG from figure 2.14 displayed in an SVG browser plug-in.

cluding a script, allowing different scripting languages to be used. The SVG specification does not make clear what scripting languages, if any, must be supported by a conformant viewer, although ECMAScript seems to be the de-facto standard. The ECMAScript language[17] is based on the common elements of JScript and Javascript from the Microsoft and Netscape web-browsers. Other languages, such as Macromedia's ActionScript are also ECMAScript compatible. While all of these scripting languages meet the ECMAScript requirements, they all have their own idiosyncrasies, particularly in regards to their object models. Another feature of SVG is that it is designed principally with the web in mind. The standard expects SVG to be embedded in web pages, and it provides built-in functionality like hyper-linking to other web resources.

Implementations

The dominant implementation of SVG already allows the embedding of SVG graphics in web pages. This mature implementation is the Adobe SVG plug-in[55]. There are versions for Macintosh, Microsoft Windows, Solaris and Linux operating systems. The Windows implementation is compliant with the Microsoft ActiveX standard, and therefore can be built into other Windows programs. Other implementations include a pure Java system for SVG rendering[20], a KDE plug-in[27], and the first native web-browser implementation in Mozilla[44]. Also, Corel are developing a "Smart Graphics" product line based on SVG[12]. This will include the Corel Smart Graphics Studio for creating SVG interfaces, and the Corel SVG Viewer for viewing SVG content in browsers or otherwise. Additionally, Microsoft's new version of their Office suite will include support for the import and export of SVG graphics through their Visio software[25]. Even with these later implementations still in development, there is clearly already an adequate base of support to make SVG a valid option for use in software visualisation.

2.5.3 The future of SVG

With the 1.0 SVG standard now a formal W3C Recommendation, new versions of the standard are already under development. SVG 1.1[62] contains no new features, but is a modularisation of SVG into "profiles". Different implementations can then support particular profiles, designed for specific tasks, such as SVG Mobile, and SVG Tiny. SVG 1.1 now has the status of W3C Proposed Recommendation, and is expected to become a formal Recommendation in the near future.

Unlike SVG 1.1, the SVG 1.2[63] standard aims to add a number of interesting features. These include automatic text wrapping, declarative drawing order, streaming, and support for XML XForms, among others. SVG 1.2 is currently a working draft.

2.5.4 Competing vector graphics formats

The web has long been dominated by *bitmap* graphic formats such as JPEGs and GIFs which are standards for compressed bitmap image data, but vector graphics are not new either. SVG has one major competitor in Macromedia Flash. Flash is the de-facto standard in web vector graphics and already has a broad development base. Macromedia provide a number of mature authoring tools for web designers, and have free viewing plug-ins for Windows, Macintosh and Linux. Macromedia flash also provides animation, interactivity and many graphics capabilities. One core difference between flash and SVG is in the file storage details. While SVG documents are stored in a plain text format (XML), Flash content is stored in a binary format. The implications of this, as well as further comparison of the formats will be included in section 5.4.1.

2.6 Evaluating SVG

Following this background, we can now move onto the evaluation of SVG as a medium for software visualisation. While SVG looks promising, at this stage it is clear that there are other technologies available which may fit VARE's requirements. Each will have its own strengths and weaknesses, and as we shall soon see, the complexity of the requirements makes it difficult to pick the best technology without careful analysis. We need to be able to evaluate SVG, along with other graphics technologies in order to find the best fit. Such an evaluation needs to be done in a consistent way to allow clear and fair comparisons. To this end, we will move forward and develop a general model for the *evaluation of computer media for software visualisation*. Not only do we need a model to make comparisons fair and easy, developing a clear model ensures that requirements are explicitly laid out. This makes a complete and thorough evaluation possible. Only once we have such a model can we then move onto evaluating SVG as a specific candidate for VARE, as well as for use in software visualisation in general.

Chapter 3

Evaluation Model

The aim of this chapter is to develop a model for evaluating software visualisation computer display media. This evaluation model will be based on existing information and software visualisation principles. While the immediate goal is to assess SVG's suitability for inclusion in the VARE architecture, developing a more generic model has two prime motivators. Firstly, SVG is not the only choice for realising visualisations in VARE. Therefore, we also want to be able to utilise our evaluation criteria on technologies other than SVG. More importantly, it is not evident that a model for the evaluation of software visualisation media has been created to date. This may also be true even for the broader area of information visualisation. As such, developing a clear and comprehensive model should be a positive contribution to the area.

For the purposes of this model, we define a software visualisation "medium" to be any technology or technologies used in the creation, deployment and display of graphical images to an end-user via a computer display. Note that "creation" here means only the specification of the graphics for display, *not* the whole process of generating graphics from a program execution. To put this in the terms of the PMV model (section 2.4.1) we are only creating a model for the evaluation of the *visualisation* component, not the *program* or *mapping* components.

3.1 A basis for evaluating software visualisation media

The problem in developing this evaluation model is that the area of software visualisation is still so new. As we are still exploring which visualisations are useful for understanding software, it is difficult to pinpoint exactly what capabilities a medium must provide. If we limit ourselves to only supporting current software visualisations, our model is unlikely to remain relevant when a visualisation is conceived which demands new capabilities of the medium. The most practical way to address this issue is to try and identify all of the ways in which information can be visually encoded. In identifying the software visualisation designers' palette we can evaluate a medium not just against what is required in software visualisations today, but also against what we can expect to be required in the near future. Fortunately,

much of this work has already been done in the more general field of information visualisation. This is briefly described in section 3.1.1.

It is also important to explore in more detail the types of visualisations that are particularly suited to software visualisations. We need to examine existing software visualisations and utilise the existing taxonomies of software visualisations which have already been developed in the literature. This is covered in section 3.1.2.

3.1.1 Information visualisation

The field of information visualisation (briefly introduced in section 2.2) has steadily developed a comprehensive analysis of how information can be best encoded in visual form. Our interest here is not with what decisions to make when constructing visualisations, but with the visual palette that should be available to visualisation designers. The basis for this work is Jacques Bertin's *Semiology of Graphics*[3] published 1967. This classic work dealt only with "that which is...on a sheet of white paper"[3](p.42) in defining the palette. While this was a reasonable limitation for the time, it obviously needs extending for our purposes. However, Berin's work provides a good starting point for the first part of our evaluation model. Building from this work, the information visualisation community have extended the palette to include a vast array of possibilities afforded by modern graphics technology. Our model incorporates these additions, drawing primarily from the summary sections of "Readings in Information Visualization: Using Vision to Think" by Mackinlay et al.[7]. In addition, our model also utilises the discussion and summaries from the text "Information Visualization: Perception for design" by Ware[69].

3.1.2 Software visualisation

Because we are evaluating the capabilities of a graphics medium, we are primarily concerned with the end product of the software visualisation process, in other words, the visual depictions of software. However, taxonomies of software visualisation have often focused on other aspects of the visualisation process. Oudshoorn et al[47] describe a number of these taxonomies: Price et al.[1] have focused their taxonomy on general features in tools used for software visualisation. Roman and Cox[51] attempted to categorise the methods of providing visualisations, while Kraemer and Stasko[28] looked in detail at the process of transformation from execution to graphical representation. Oudshoorn et al. then go on to discuss their own taxonomy based on what information travels from the underlying program execution through to visualisation.

None of these really serve our requirements for a taxonomy of the types of graphical representation needed for software visualisation. However, Oudshoorn et al. do briefly mention what would be the basis for such a taxonomy. They break down the types of "data representations" for software visualisation into three "well-known types".

- Graph-based displays

  Graph-based displays are what Ware[69](p.222) calls node-link diagrams. We shall also use this term. They are built from nodes which represent entities and links between them which represent various relationships. Different node attributes (shape, colour etc.) are used to encode information about the represented entities, and similarly for the links. The most common group of these software visualisations must be those from UML. An example of UML's collaboration diagram was shown in figure 2.9. Collaboration diagrams are a classic example of node-link diagrams with their boxes for objects, and lines for method-calls.

- Statistics-based displays

  Statistics-based displays move into the area of scientific visualisation. Such displays can use aggregations of data to draw statistical graphs, or can display data in a massed form by encoding it visually for rapid assimilation by users. A host of examples of statistical software visualisations can be found in De Pauw et al.'s work in creating an architecture for visualising program behaviour[49]. Another example can be seen in figure 3.1 by Jerding et al[26]. This example, called the "Execution Mural" shows graphically the entire record of messages passed in a program execution. Here the colours and spatial mapping may help a developer see patterns emerging.

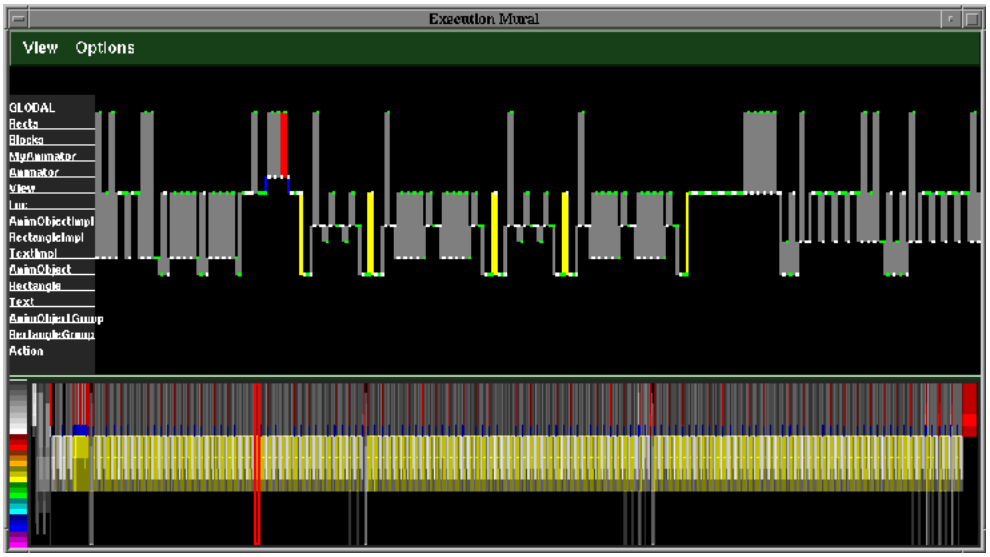


Figure 3.1: A visualisation showing messages passing between objects in an executing program as a "mural". The bottom half shows the message stream from the entire program, while the top half shows a detail.

- Source-code-related displays

  Source-code-related displays show source code in a more visually accessible form. This can include a zoomed out source code view with additional information encoding such as Eick et al.'s SeeSoft[19] shown in figure 2.2, or in providing linking from a UML class

diagram to the associated source code as in figure 3.2. I created this second figure in SVG for the evaluation, and it is explained further in section 4.2.2.

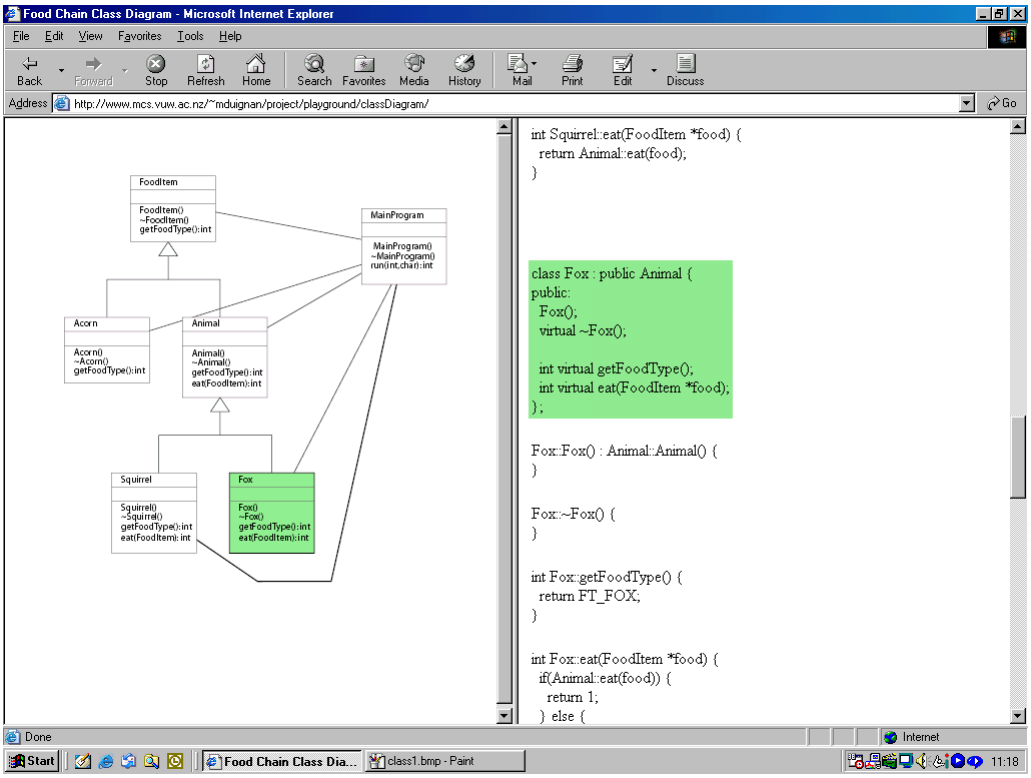


Figure 3.2: A UML class diagram I created in SVG and HTML which displays and highlights associated source code.

3.1.3 Ease of programming

A model for the evaluation of a medium for software visualisation needs to take into account how the visualisations 'get into' the medium. If visualisations are created directly by a human user there will need to be good tools for them to use. If the visualisations are to be created programmatically (as in VARE) the medium and the libraries that facilitate diagram creation need to provide the capabilities in a logical and intuitive manner.

3.1.4 Limitations

Vitally important in any model for evaluation is identifying what is being left out. The following areas will not be included in the model:

- We are not dealing with senses outside of vision. While the use of touch, smell and sound is potentially interesting, these fall outside of the scope of this model. However this model could be extended at some later time to include them.
- We are only dealing with 2D computer display technology. The main reason is that this is the equipment that has become cheap and accessible, and is therefore the most

useful as a vehicle for software visualisation at this time. Again, expanding the outlook of this model to wider possibilities would be interesting, but is left as future work.

- Because we are not examining how to best create visualisations, we are not directly including end-user usability in our model. However, end-user usability does make some requirements on the technology (as opposed to what we do with it) so it will be considered where appropriate.

- Finally, other than general future proofing (through a focus on the requirements for basic information visualisation), we are not trying to predict where software visualisation will be heading in the future. It is quite possible that despite my best intentions, radically new visualisations will be created that require capabilities not included in this model.

3.2 Model framework

The framework for this model is based on the discussion above and takes the following basic structure:

- Firstly we explore the basic capabilities which enable the encoding of information in a visual medium (Section 3.3). This includes graphics, interaction and performance requirements based on the work of Bertin and Mackinlay et al..

- Next we look at higher-level issues — concentrating on how well the medium supports the things we want to do, rather than just providing the basic capacity to do them (Section 3.4). This includes looking at how well the medium integrates with other technologies we wish to use, and what we shall call "programmer usability".

- The next step is to look at the various types of software visualisations identified by Oudshoorn et al. discussed in section 3.1.2. These need to be fleshed out into bench marks for inclusion in the evaluation model (Section 3.4.3).

For each of these broad categories of analysis, there will be a number of capabilities that our model will identify. To use the model effectively, it is necessary to describe how well the medium meets each criteria, rather then simply giving a yes or no answer. This is because different media will support each capability differently, and to a greater or lesser extent.

3.2.1 Qualitative evaluation

Evaluation in the model will be qualitative in nature. The reason for this is that many of the capabilities we are looking for in a software visualisation can not be easily broken down into perfectly measurable units. For example, we may want our medium to support us in adding textures to areas of a graphic. But the realised support for this could vary immensely.

While some media may support this directly, others may allow this only through some other (but more awkward) means. A programmer dealing with a medium may have to calculate every pixel values for an entire surface for texturing, while in another medium simply specify a texture from a predetermined list. Yet another medium may allow for arbitrary textures to be imported for display. It is often possible to get a technology to do things that it was not designed for, but the question to ask here is: how well does it support these particular functions? While it might be possible to come up with the ultimate quantitative model, this would be a difficult task, and it would be forced to ignore many of the quantitative facets of capability that we are interested in.

3.3 Basic information visualisation capabilities

This section is concerned with describing the visualisation palette needed for software visualisation design by utilising the work of the information visualisation community, in particular the work of Bertin and Mackinlay et al..

3.3.1 Graphical capability

Graphical properties are the most important properties that a medium for software visualisation must support. Indeed, the term "software *visualisation*" itself makes this an obvious focus. Graphical media are generally described in terms of the "spatial substrate", "marks", and the "marks' graphical properties"[33]. The spatial substrate describes layout, while marks and their graphical properties describe the visible elements which are layered upon this. The importance of this is emphasized by the statement that "...while other properties...are possible, [...] most visualizations will probably continue to be made from this basic set."[7](p.26).

The spatial substrate

The spatial substrate is a term used to describe the underlying *spatial* characteristics of a graphic. The spatial substrate can be organized in various ways to give differing meanings to the position of marks. A display medium's support for different aspects of the spatial substrate can be broken down as follows:

- Dimensional support

  While we have limited our model to include only 2D media, even these can allow differing support for the illusion of additional dimensions. A medium may even be able to represent 4+ Dimensions through the use of clever built-in techniques[1]. Some simple

[1]Such techniques are often employed in Data Warehousing for the representation of data with multiple dimensions.

techniques for 3D display include stereo-scopic depth, kinetic depth (creating the effect of a rotating 3D object) and shading.

Example use: 3D can be used in visualisation to support focus + context. UML type class diagrams have been modeled in 3D[16].

- Axis folding

  As implied in *dimensional support* above, a 2D medium may have inbuilt support for additional dimensions — and hence additional axes. However, with limited room in the dimensions of a space it is often useful to "fold" dimensions so they fit in an allocated area.

  Example use: The classic software visualisation "SeeSoft"[19] visualises lines of code in columns with a zoomed out effect. When displaying files with large numbers of lines it breaks the files into multiple columns. Each column is an axis which is "folded" at the breaks.

- Axis types

  The traditional straight axes used in line and bar graphs are not the only option for display of ordered data. Other axis types include curved or circular axes, as well as axes along arbitrary paths.

  Example use: Data displayed in the pie graph form is one obvious example. This could be used in displaying program statistics.

- Axis distortion

  Distorting axes can be very useful when implemented correctly. Distorting an axis involves modifying the regular intervals along its length to create a desired effect. The result is similar to a ruler with the marked measurements having been "squashed" or "stretched".

  Example use: Distorting an axis is often used to create focus plus context views. Focus plus context provides a detailed view of some information, while still providing a contextual view of the surrounding information. One example from Information Visualisation is the "perspective wall"[7] where a two dimensional space is projected as a bent wall in a three dimensional space. While this relies on three dimensions, a similar effect could be created just through squashing context information on an axis in two dimensions. This could be used in very large UML sequence diagrams.

- Viewpoint control by system

  Ware asserts that "moving the viewpoint in a visualization can function as a form of narrative control"[69](p.327). Note that here we are talking about the visualisation controlling the viewpoint, not the user controlling the viewpoint as in interaction (section 3.3.2).

  Example use: An animated sequence diagram would need to pan and scroll its viewpoint when the action moved out of the current viewpoint.

- Recursion

  It is possible to allow the repeated subdivision of space providing a recursive space[7](p.28).

  Example use: A software visualisation could be created that contained a host of visualisations inside itself. Users could pick a visualisation by zooming in. Having support for recursion of space would allow each of these sub-visualisations to have their own spatial coordinates etc.

Marks and their properties (retinal encoding)

Marks are what we directly experience when viewing a visualisation. Therefore, the types of marks we can make, and the properties that they can have are vital capabilities for a software visualisation medium. The capacities itemized here largely take the form of vector graphical capabilities. This is due to the assumption that in creating these visualisations we want to think about it as 'making marks' on a space. The act of making a mark is essentially a vector based notion — regardless of how it is implemented by the system. What this comes down to is that when we are creating the visualisation in the medium we do this through vector based operations (e.g. make green line between (0,4) and (3,12)).

For each of the capabilities identified below the questions to ask of the medium are: can we specify marks in this way, and is this information retained at display time? For example, a library for creating bitmap graphics may allow us to specify marks in this way but these marks will then lose the identity they would have kept had they been represented as vector graphics. This will become important for interaction (3.3.2), integration (3.4.1) and higher-level (3.4.2) capabilities.

- Size

  Do we have control of the size of the marks we make? Can we scale the marks we make?

- Colour

  Can we control value, hue and saturation?

  Note: For colour coding information we can use usefully only about eight different colours[69](p.194) although having a continuous range is necessary for colour gradients and other effects.

- Orientation

  Can we rotate marks through 360°?

  Note: For coding discrete data through rotation we can usefully make use of only four rotations[69](p.195) as smaller rotations are difficult to distinguish.

- Shape

  Can we specify lines paths and curvature?

  What is the expressiveness of the way we can define them?

- Points, lines and areas

  Can we specify points, lines and areas?

- Filter effects

  Can a particular area be altered through filtering effects? Effects could include resolution and crispness.

- Transparency

  Can we control a see-through effect on particular marks and areas?

Temporal encoding

Temporal encoding is simply changing the visualisation over time to communicate additional information.

> "The use of simple motion can powerfully express certain kinds of relationships in data [and the] animation of abstract shapes can significantly extend the vocabulary of things that can be conveyed naturally beyond what is possible with a static diagram."[69](p.239)

- Encoding time

  Can graphics be easily changed to show the passage of time?

- Encoding identity

  Can animations be specified on particular entities for the smooth animation necessary to show identity?

- Variation in retinal encodings

  For full expressive control there needs to be control of the value of all retinal encodings (section 3.3.1) over time.

3.3.2 Interaction

Interaction is an essential part of many types of visualisation.

> "In some ways, a visualization can be considered an *internal* interface in a problem-solving system that has both human and computer components. A visualization can be the interface to a complex computer-based information system that supports *data gathering and data analysis*."[69](p.335)

In this respect visualisation becomes part of an active process controlled by the human user. The ideal way to integrate visualisation into this active process is to make it interactive. This clearly applies to software visualisation as documentation. In the VARE architecture the user is engaged in *data gathering and data analysis*. This is where the area of Human Computer Interaction (HCI) with its concern for usability becomes an important influence. The discussion here does not consider what is good for the usability of a visualisation, but rather, identifies the full range capabilities that are required to make it possible to implement a system with good usability. Additionally, while many of the interaction capabilities could be side-stepped in the VARE architecture by resorting to regeneration of the visualisations at the server side, there are serious consequences to this. The central problem is the time delay caused by the latency of the network — particularly in a web-based system. If the user has to wait too long for feedback this cripples the effectiveness of the user exploring interaction possibilities. The user is essentially penalised for exploring the visualisation and may find this unacceptable. Also, an action and an associated event need to occur within a 0.1 second time frame to convey cause and effect effectively[7](p.231). This effect is destroyed by the delays introduced in client/server web-based systems. The consequence is that the visualisation medium (as deployed to the client) needs to provide as full a range of interaction capabilities as is possible.

The following capabilities are important for interaction in visualisation systems and should therefore be part of our evaluation criteria.

- Graphic malleability

  Interaction demands that the graphic representation can be changed at runtime in response to user actions. Ideally we should be able to change the graphic arbitrarily at runtime. In a less optimal system, potential graphic changes might have to be built into the representation before deployment. In the worst case the medium might be completely unchangeable once created.

  Example use: A class diagram could be augmented to link to source code. Highlighting could be used to make a link between currently viewed code and the corresponding class in the diagram. This would require being able to change the colour of classes in the diagram at runtime in response to the user's actions.

- Events

  Interaction with a visualisation is limited by the types of events it can recognise. Possible events include mouse events (mouse over, mouse click, mouse off etc.) keyboard events, timing events and process events (for interprocess communication). Limiting the types of events the visualisation can recognise limits the potential usability of the system (e.g. a mouse-over might be more intuitive then having to use the keyboard).

  Example use: Brushing is a common visualisation mechanism and could be used as follows: Source code structure could be visualised in the same zoomed out manner as

SeeSoft[19]. When the mouse passes over (or 'brushes') a method declaration, all of the calls to this method in the zoomed out source code could be highlighted.

- Computation

  When the visualisation receives one of the above events it needs to be able to respond in an appropriate fashion. This will often require some level of computation on the system's part. Does the visualisation medium allow us to compute everything that we need to?

  In a typical implementation we would expect to have the capability for condition testing and loops as well as variable storage and manipulation.

  Example use: Again using the SeeSoft example, functionality could be added to allow the user to enter a string of text to be searched for. Hits would be highlighted in the visualisation. Of course the medium would need to be able to conduct the equality testing and string operations for the search, as well as be able to handle and respond to malformed user input.

- User notation

  To support data analysis a visualisation system could allow the user to annotate visualisations for their own reference. This could be supported through implementation of the other interaction capabilities listed here, but it could also be implemented natively in the medium.

  Example use: A user could annotate a UML class diagram with personal notes.

- View refinement/navigation

  Giving the user control of the visualisation's viewpoint empowers them to uncover and follow up information that they are interested in. View refinement and navigation becomes important when there is information extended out of the current viewpoint[69](p.343) and this will always be the case when zooming is permitted. Rapid and easy to invoke zooming is a way to provide focus + context[2]. Allowing the user to control the clipping (mask) of the view-port can give additional flexibility.

  If the medium implements viewpoint control natively it may support various spatial navigation metaphors which could impact on usability. These include 'World-in-hand', 'Eyeball-in-hand', 'Walking', and 'Flying'[69](p.346).

  Example use: UML diagrams can become very large. Enabling the user to pan and zoom is therefore very important.

- Information hiding

  Enabling the user to hide information they are not interested in allows them to avoid information overload. This also allows the user to uncover increasing levels of detail as they are ready or require it.

Note: We are not referring to information hiding from Object Oriented design. Rather the literal "hiding" of information in the visualisation.

Example use: A UML sequence diagram could be collapsible and have method activations and their consequences shown only as the user clicked on them.

- Time control

  If the visualisation has a temporal encoding component (see section 3.3.1) it might be important for users to control the current position in time. This could include "stop", "start", "restart", "rewind", "fast-forward" etc.

  Example use: The VARE architecture hopes to use a video playback metaphor for visualisations with animation[4]. Giving the user control of the animation's current position would be vital for this.

3.3.3 Performance

In interaction (section 3.3.2) the impact of time delays in interaction on usability was briefly discussed. Clearly the performance characteristics of the medium will have a direct impact on this.

- Scalability

  Does the medium have inherent scalability problems? Even if the performance is acceptable for small visualisations, it may drastically degenerate as it grows in size.

- Current implementations

  It is possible that performance issues may only be the result of current implementations rather then being inherent in the medium itself. How good is the performance of the current implementations?

3.4 Software visualisation specific and higher-level capabilities

This section is concerned with software visualisation specific, and higher-level, capabilities. While the software visualisation specific capabilities described here are based on established models of software visualisation, the higher-level capabilities are a new area which I have developed.

3.4.1 Integration

No computer technology is isolated. An operating system is only as useful as the applications that it hosts, and a web browser is not much good without the resource of web pages and web

servers for it to interact with. Many technologies may seem simple or limited, but because they integrate closely with other tools, the synthesis of their features make them worth much more then their individual potential might suggest. The same is true for a visualisation medium.

We need to ask, what does this medium gain from its integration with other technologies. This raises a number of issues:

- Creation mechanism

  The visualisations could be created by a human user "drawing" the diagram with the aid of a tool, or through programmatic generation. For both these cases, different tools/technologies will provide differing advantages and disadvantages. What tools/technologies are leveraged by the medium for creation?

  Example use: The VARE architecture supports the automatic creation of software visualisations from (executing) program code. It is also aiming to allow human augmentation of generated visualisations for additional documentation.

- Deployment

  If a visualisation medium is difficult to deliver to potential users, its usefulness will be severely limited. How does the visualisation get from the generating process or person to the end user? Also, one way of conducting software visualisations is to have the graphical representations appearing in real time as the program runs. This way, users can interact with the program and see the visualisations of the internals as they play out. This requires a more sophisticated deployment mechanism.

  Example use: This question is particularly important for use in the VARE architecture. The VARE architecture aims to deliver visualisations over the Internet using standard web and Internet technologies. Obviously, the easier the medium integrates with these technologies the better.

- Linkages with other technologies at display time

  The medium may utilise other technologies to provide additional functionality. Alternatively the medium may be a completely stand alone technology. Linkages between the technologies need to be included in the evaluation to ensure that a medium is not penalised for following modular design principles — providing only one part of the functionality, but doing so in a way which can easily be integrated with complementing tools.

  Example use: If further documentation is available outside of the visualisation this could be made available by utilising existing HTML technology. Clicking on a class in a class diagram could cue a web browser to load and display the associated documentation or source code.

- View coordination with other visualisation media

If different media have different strengths and weaknesses we may want to use more then one visualisation medium in conjunction. However, a visualisation technique that utilises multiple complementary views[48] of underlying information requires view co-ordination. Can the medium under investigation integrate with other media for view coordination?

Example use: If one view of a multiple view software visualisation was being displayed using Macromedia Flash, and another in a Java Applet there would need to be communication hooks between the two to support maintaining consistent views.

3.4.2 Higher-level capabilities

While the areas of information and software visualisation seem to be primarily concerned with what makes good visualisations and the underlying technical design of visualisation architectures, there is another area which we need to examine. In evaluating a medium we need to be very mindful of what I will call "programmer usability". The fact is that many media will support our basic information visualisation capabilities (section 3.3). However, a medium's effectiveness is not primarily determined by what it can do in theory, but rather how easily you can get it to do what you intend to do. If one medium requires ten times the development effort of another we need to be aware of it. Higher level capabilities aid programmer usability by integrating useful operations into the technology itself.

- Higher order graphics

  One set of higher-level capabilities is to provide inbuilt support for creating common shapes, dealing with text, creating layering and filtering effects, textures and other common requirements.

  Example use: A 3D object model could have lighting and shadow effects to help give depth cues. If these were not provided by the medium the effects would have to be programmed by the visualisation designer.

- Data/display independence

  Being able to keep underlying data independent of the visual display is a common design practice. The Model View Controller pattern[6] and the software visualisation specific Program Mapping View model[45] are both variations on this principle which helps isolate the ideal representation of the data from the current implementation of the display of that data. But in software visualisation, this data/display independence is not just important over the whole program to visualisation conversion process, but also in the visualisation medium itself. The reason for this is for support of interactivity. Once a visualisation has been created we need to be able to change the details of how it is displayed without altering the underlying structure of the visualisation.

  Example use: In a UML sequence diagram with collapsible method activation boxes the visualisation needs to track the un-collapsed size of boxes when they are displayed

as collapsed. This information needs to be stored somewhere. Another example is in trying to avoid "conjunction searches". Conjunction searches are where the user visually searches a graphic for elements with two or more particular properties[69](p.169). For example, a software visualisation could show a zoomed out depiction of every current object in a running program (typically in the thousands even with a simple Java program). Object size could be visually coded in gray-scale shade and object type as shape. Looking for round black objects is made very difficult if there are numbers of other black objects and other round objects. If upon deployment, the data-to-visual attribute mapping is still flexible in the visualisation medium, users can avoid this problem through interaction. This requires some level of independence between the data and the display.

- Referencable entities/objects

  This is related to the visualisation being kept in a vector based form (see section 2.5.1). For implementing interaction it is very important that graphical elements can be referenced so modifications can be made.

  Example use: In the example of a UML class diagram linked to source code or documentation, it is necessary to highlight the class currently being viewed. This requires being able to refer to the graphic of a class in order to change its colour.

- Layout constraints

  Support for layout constraints can make the creation of visualisations radically simpler. This allows the programmer to specify what is important in layout and by implication what is not. This means that the programmer has drastically less work to do in calculating the effects of any graphical change during interaction. Being able to implement visualisations using simple layout constraints could hugely reduce development times. Layout constraints make it easier to provide semantic preserving manipulation[56] which allows diagrams to retain their meaning as they are modified.

  Example use: In a UML collaboration diagram the user may wish to move objects around for a better layout. This is particularly true with diagrams created by automatic layout algorithms which are often non optimal[15]. An intuitive way to support this is to allow the user to click and drag the object to a new location. The effects of this should be followed from layout rules rather then hard coding by the programmer. For example, the object may be connected to any number of other objects by lines showing relationships. All of these lines should be updated to point to the new location of the object. Another example would be with a collapsible UML sequence diagram. When part of it is expanded, the lower segments need to be moved down. This too could be automated with layout constraints.

  Note: Layout constraints can also enable other useful capabilities such as semantic zooming and differential scaling[56] — but we will not describe all of these here.

- Structure

When having to create and manipulate the graphical elements of a visualisation, it is important that the elements can be grouped and structured as the programmer sees fit. If the programmer can create underlying structure for graphical elements appropriate to the demands of their application, manipulating the visualisation should be easier.

Example use: Many software visualisations contain arrows. An arrow might be made up of a number of lines and shapes in its implementation. If these elements can be grouped together they can be manipulated in a more logical form.

This area of higher-level capabilities (or programmer usability) in graphical representation does not seem to have received the same amount of literature backing that we see for other areas of information and software visualisation. Apart from ideas found in a paper on the weaknesses of SVG[56], the above capabilities are largely based on the author's personal experience in carrying out this current evaluation. The end result of this is that this part of the model is relatively immature as it is not based on the years of research by many parties as in the broader visualisation area. There could be other higher-level capabilities such as causality of events which might be important in creating software visualisations. Developing this further will have to be left as future work, although providing this initial discussion as an first effort is important.

3.4.3 Support for current software visualisations

In addition to the capabilities which have been identified so far in this chapter, a software visualisation medium needs to be assessed against the current categories of software visualisation. In section 3.1.2 we identified three main types of software visualisations. We therefore base our examination for support of current visualisations on instances of these three types. It is likely that as the area of software visualisation moves forward the visualisations I have chosen will need to be updated.

- Node-link displays seem to be the predominantly used type (expressly for object-oriented programs) and so are the most important to include in an evaluation. UML offers a number of node-link variations. The two most obvious and common are the Collaboration diagram and the Class diagram. Sequence diagrams are similar to the basic node-link type. Since the grammar of UML is standardised and largely understood these diagrams are good choices for inclusion as tests in the model.

- What we have been calling statistics-based displays seem to fall into the two categories of aggregated data, and raw data display. Aggregated data displays (like pie and bar graphs) hardly seem likely to tax any medium under investigation. For this reason, we will specify a raw data type display. Examination under this model should include the creation of a display similar to the message mural shown in figure 3.1.

- Source-code-related displays are the third common type of display. For evaluating the medium against this type we will use a simple source-code browser driven by a UML

class diagram. Activating the image of a class in the class diagram will locate and highlight the associated source code. The image of the class should also be highlighted to emphasize the currently viewed class and the relation of it to the code in question.

Chapter 4

Exploring SVG

Having developed an evaluation model, we can now proceed to steps two and three of our methodology: Examining SVG's capabilities, and constructing SVG software visualisations. This examination takes a number of forms. These include reading SVG's formal specification, doing tutorials, examining existing examples, as well as creating a number of SVG software visualisations — through building programs to create SVG, authoring SVG by hand, and with graphical user interface drawing tools.

4.1 Learning SVG

The first step to being able to evaluate SVG is to become proficient with the details of the technology. This can be accomplished through using web resources as discussed in the following sections.

4.1.1 The SVG specification

SVG is a publicly documented standard created by the World Wide Web Consortium (W3C). As such, an important part of understanding the capabilities of SVG is reading the standards document[61]. While the document is large, it is well written and easy to read, as well as containing illustrative examples. While reading the document is time consuming, it provides the necessary knowledge to be able to assess SVG against the model I have developed.

4.1.2 On-line tutorials

On-line tutorials provide a good foundation for a more practical knowledge of a technology. Adobe systems provide a large range of SVG tutorials that range from the basics of creating shapes and text, through to creating dynamic SVG with scripting and interactivity. These seem to be the best tutorials available. This is probably explained by the fact that they are created by the same company who have built the leading and only fully functionally SVG viewer currently available.

4.1.3 On-line examples

While SVG is still a new standard (Version 1 was finalised only in September 2001) there are a number of highly illustrative examples on the web. Again, the prime source has been Adobe Systems who have provided a number of interesting examples[54] to motivate developers to pick up SVG. Because deploying SVG graphics requires that the source code is sent to the client for rendering, it is possible to discover how examples are created by examining the details. Reading the source code of a number of more complex examples is an important part of the learning process. Such examination reveals tricks and tips for the creation of complex and dynamic SVG content. I identified five examples of particular interest. They are briefly described here, and their implications for our evaluation will be discussed in chapter 5.

Chart and graph demo

This example allows the user to interactively enter data and have it instantly added to a number of SVG graphs on display. Figure 4.1 shows a bar graph. Users can add data through the HTML form below the embedded SVG graphic. This form cues the appropriate script function inside the SVG to draw the new column in the graph.

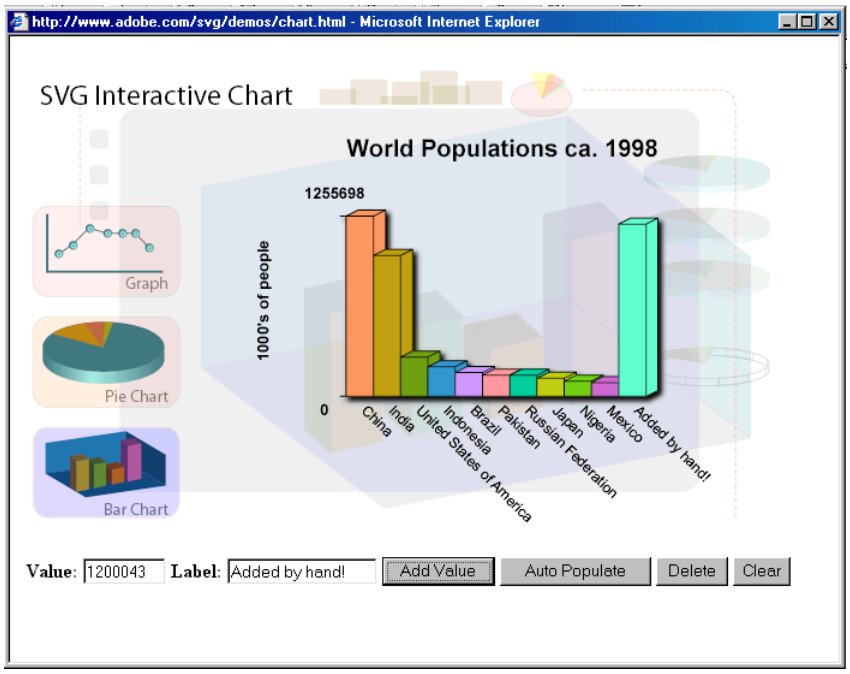


Figure 4.1: An interactive graph in SVG. (Adobe Systems)

SVG draw demo

This impressive example provides a simple vector drawing program implemented in SVG through scripting. Users can add various coloured shapes and text and then save the created

diagram as an SVG document for later use. The implementation modifies the graphic through the XML DOM. An example session is shown in figure 4.2.

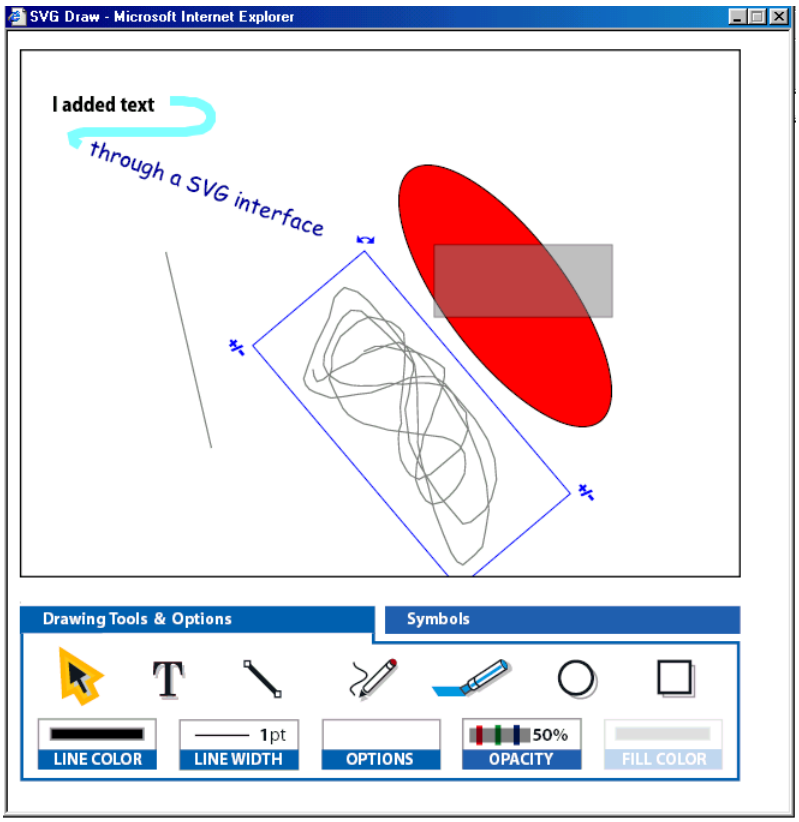


Figure 4.2: A vector based drawing program implemented entirely in SVG with ECMAScript. (Adobe Systems)

Chemical Markup Language demo

The Chemical Markup Language is another XML language, this time for the specification of chemical structures. The beginning of a Chemical Markup document is shown in the bottom half of figure 4.3. This SVG example demonstrates a number of interesting capabilities. It utilises XSLT for translating the Chemical Markup into SVG graphics. The graphical *Visualisation* of the chemical structure is a 3D representation with a shadowing effect. The user can spin the representation with a click and drag motion of the mouse. Impressively, this is all achieved in SVG's 2D environment. This is facilitated by tracking the entire model in ECMAScript, while only using SVG as a raw display medium. The script makes the calculations for rendering the 3D model on SVG's 2D plane. Figure 4.3 shows the end result.

Theater seat booking demo

This example is a prototype of a web ticking system, powered by a back-end database. The user is presented with a graphical depiction of a venue for a performance which displays

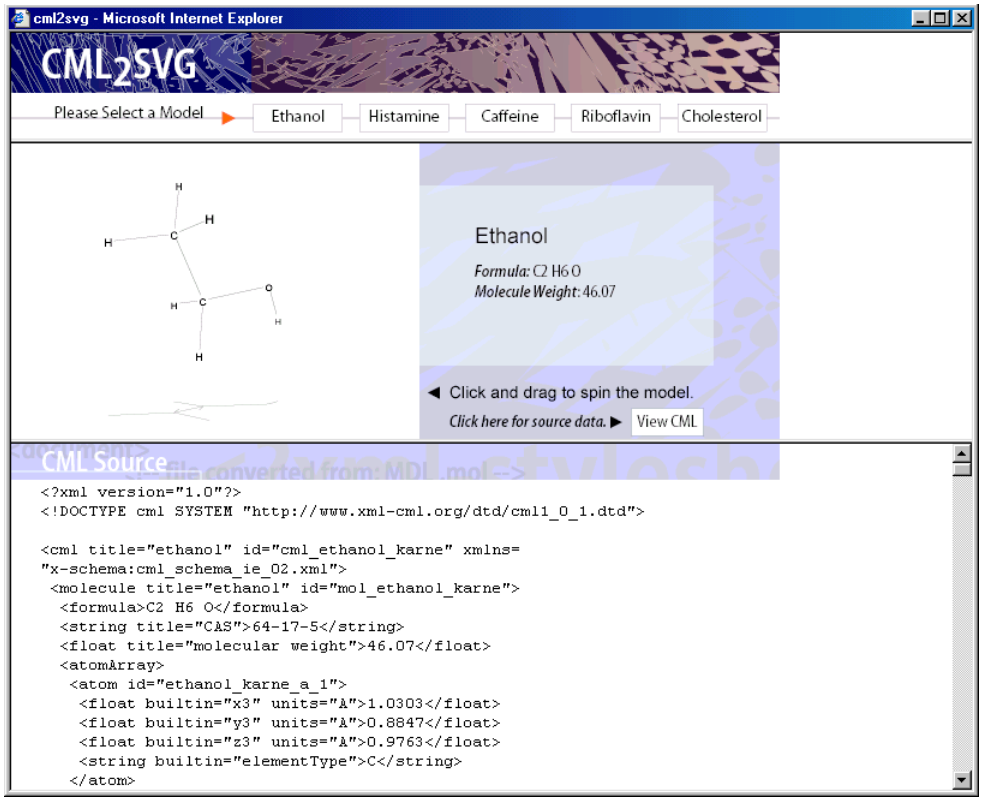


Figure 4.3: A scientific visualisation of a chemical implemented in SVG. The visualisations were created with XSLT from the XML source shown in the lower half. (Adobe Systems)

available seats (as determined by the database on the web server). Users can click on seats to book them and after selecting "buy", can enter credit card details through an integrated HTML web page. All of this is mediated by Java Servlets on the web server which both create the representation of the venue and carry out bookings via the database. The SVG user interface is shown in figure 4.4. The perspective effect is created through traditional art techniques. Three different chair symbols are specified in the SVG, each of which depict a chair on a different angle.

Apache Batik project UML class diagram

The final example is not from Adobe. This example from the Objects By Design site[14] shows UML class diagrams for the the Apache Batik SVG project[20]. This large class diagram was created from a CASE[1] tool and exported to Adobe Illustrator for conversion to SVG. Figure 4.5 shows the entire diagram zoomed out while figure 4.6 shows a zoomed-in detail.

[1]CASE stands for Computer Aided Software Engineering.

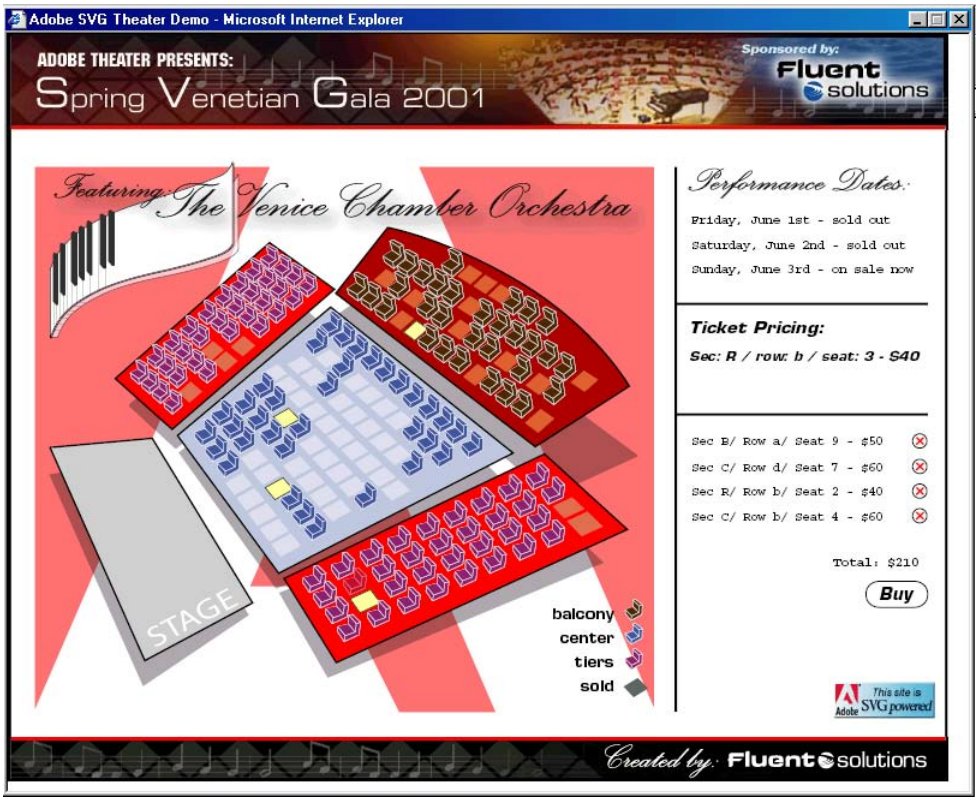


Figure 4.4: This is the graphical front end to an on-line ticket sale system with a back-end database. (Adobe Systems)

4.2 Constructing software visualisations with SVG

After having learnt SVG and discovering some of its more advanced capabilities, the task is to apply this to examples of software visualisation. The examples I chose to build were selected on the basis of fulfilling the construction requirements of our model described in section 3.4.3. The visualisations described below play the following specific roles in meeting the model's testing requirements: The UML diagrams are both variations on node-link diagrams. The message mural inspired diagram fills the statistical diagram role. The class diagram code browser is used as the code-related diagram example. In addition, all of the visualisations described below were used to further explore SVG's capabilities for use as a software visualisation medium.

4.2.1 UML visualisations

While there are many types of UML diagrams, I created only a select few for experimentation. The reason for this is that there is little benefit to creating a large number of similar types of diagrams (as UML diagrams mostly are) because most of the insight comes from a limited number of key examples. The following are brief descriptions of the UML diagrams that I implemented in SVG.

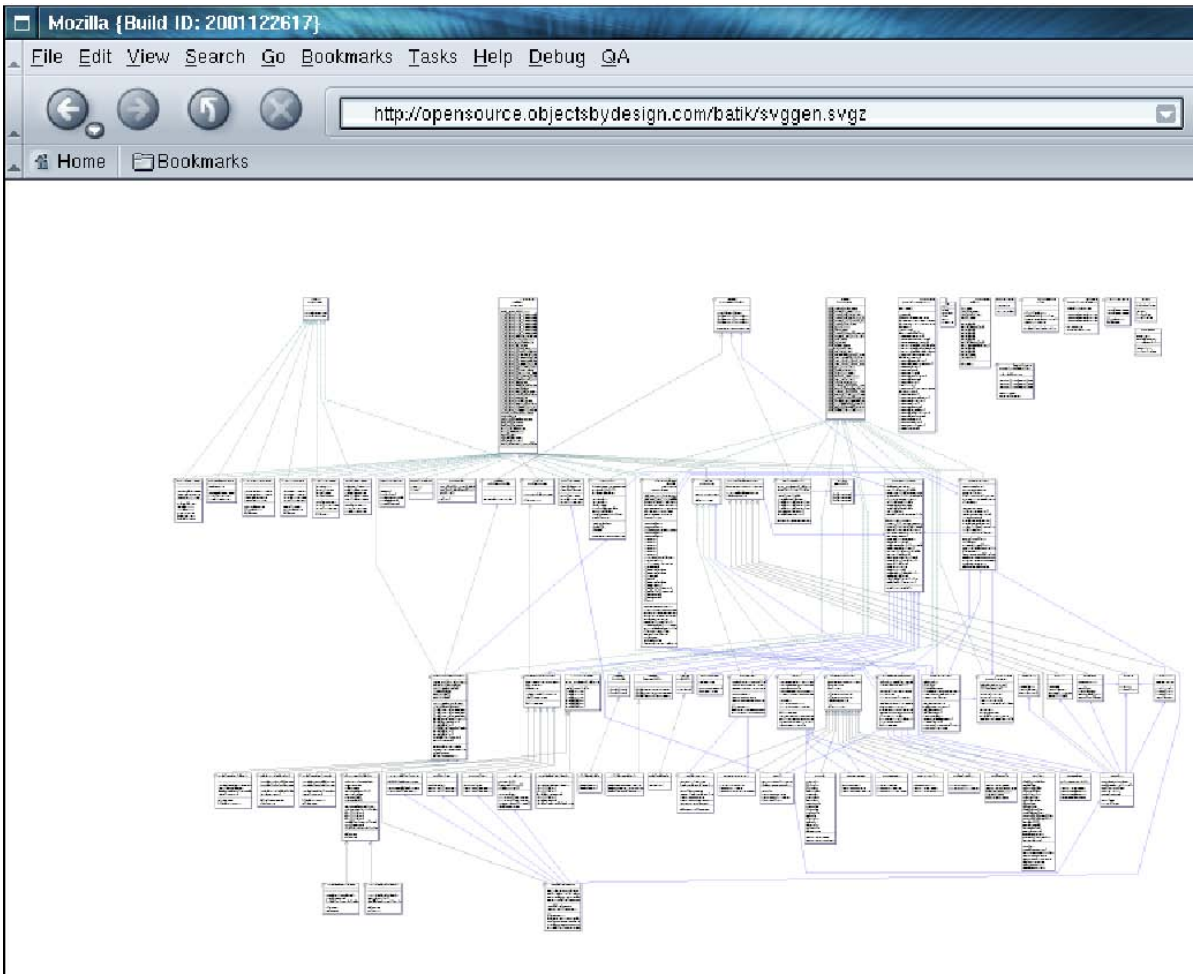


Figure 4.5: A UML class diagram of the Apache Batik software in SVG. (Objects By Design)

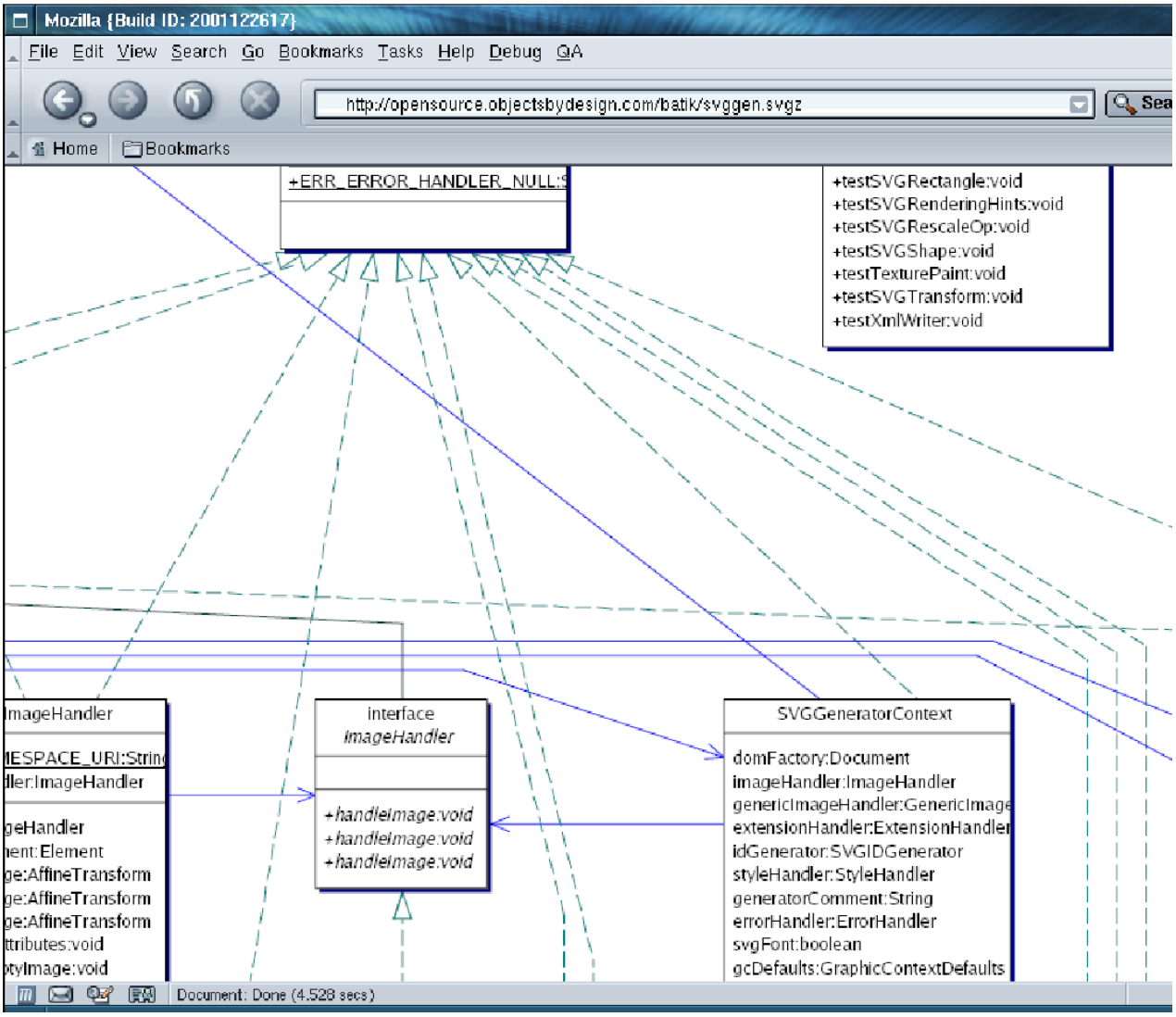


Figure 4.6: Detail from A UML class diagram of the Apache Batik software in SVG. (Objects By Design)

UML class diagrams

The Batik class diagram example is an impressive display that SVG is capable of this sort of software visualisations. However, I created a simple class diagram from scratch to utilise the learning experience of the construction process, as well as for extension as explained in section 4.2.2.

Figure 4.7 shows the simple UML class diagram implemented in SVG for this project. I created this diagram in the vector graphics program Adobe Illustrator 5. The whole diagram was created using simple drawing tools. The only step required to turn this into SVG is to choose File — Export from the menu, and then select SVG as the file format. The resulting document is then ready to be deployed on the web.

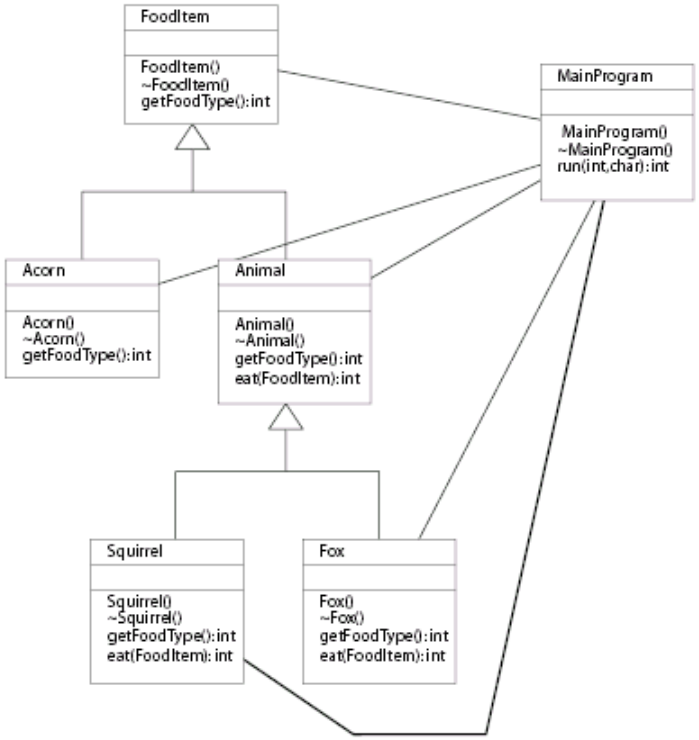


Figure 4.7: A simple UML Class diagram I implemented in SVG.

UML sequence diagrams

The second SVG example I created was a simple sequence diagram. The SVG code for this was written by hand. This example contains only the bare bones of a sequence diagram. However, creating it was tedious enough to establish the difficulty of using such a method. The diagram can be seen in figure 4.8. The method-call and object names were put in using the text element as follows:

```
<text
x="10" y="30" >Some text to display</text>
```

The arrow is specified as a symbol once and used repeatedly with the `use` element in a similar manner to that shown in figure 2.14. The stick figure is grouped together in a `g` element, as are all of the object boxes along the top. This allows a single style attribute to be used for all of the sub-elements of the group. For example, the outline thickness of all of the object boxes could be changed to 15 units by adding `stroke-width:15` to the appropriate `g` element's style attribute. While these conveniences were welcome, creating a visual image by specifying exact coordinates was a monotonous chore. After having used visual drawing tools for so long, I found that creating an image by coding was an awkward experience.

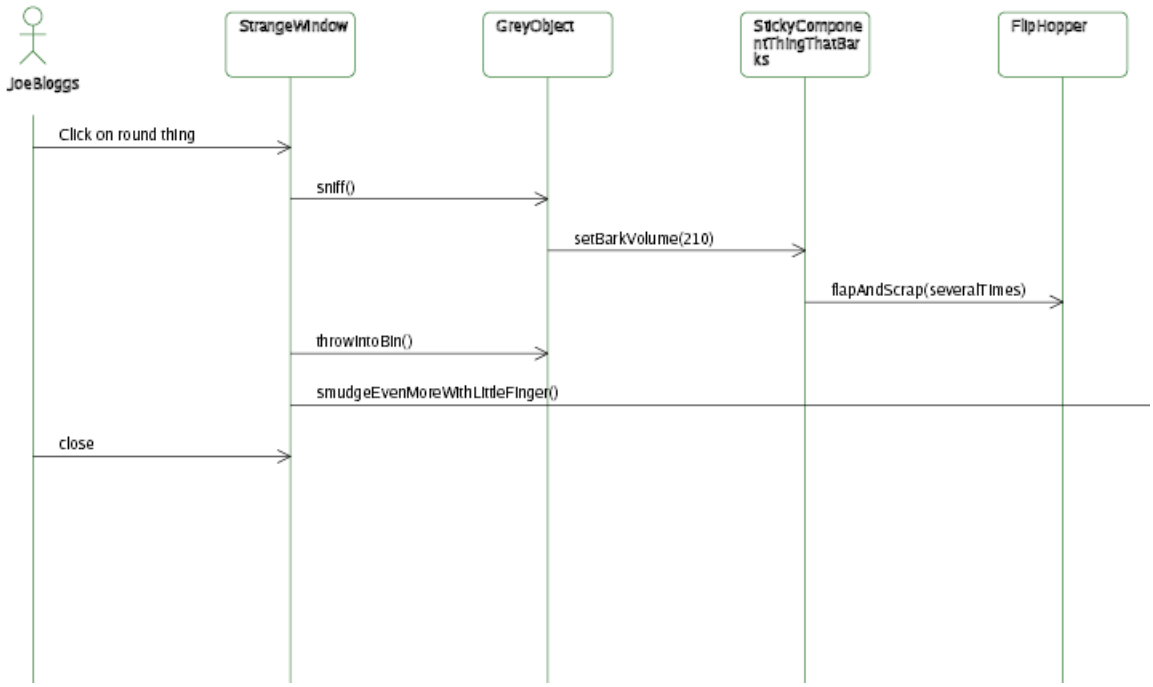


Figure 4.8: A UML sequence diagram I implemented in SVG.

4.2.2 Extended UML visualisations

To explore the interactive potential of SVG as a medium for software visualisation, it was useful to extend some UML diagrams to add interaction.

Code linked collaboration diagram

As an example of a code-based software visualisation in SVG, I developed a prototype class diagram driven code browser. This was mentioned in section 3.1.2 and can be seen again in figure 4.9. The prototype is relatively trivial in that it only contains a model consisting of six classes with a small single file of source code. However, it illustrates the capabilities of SVG to good effect.

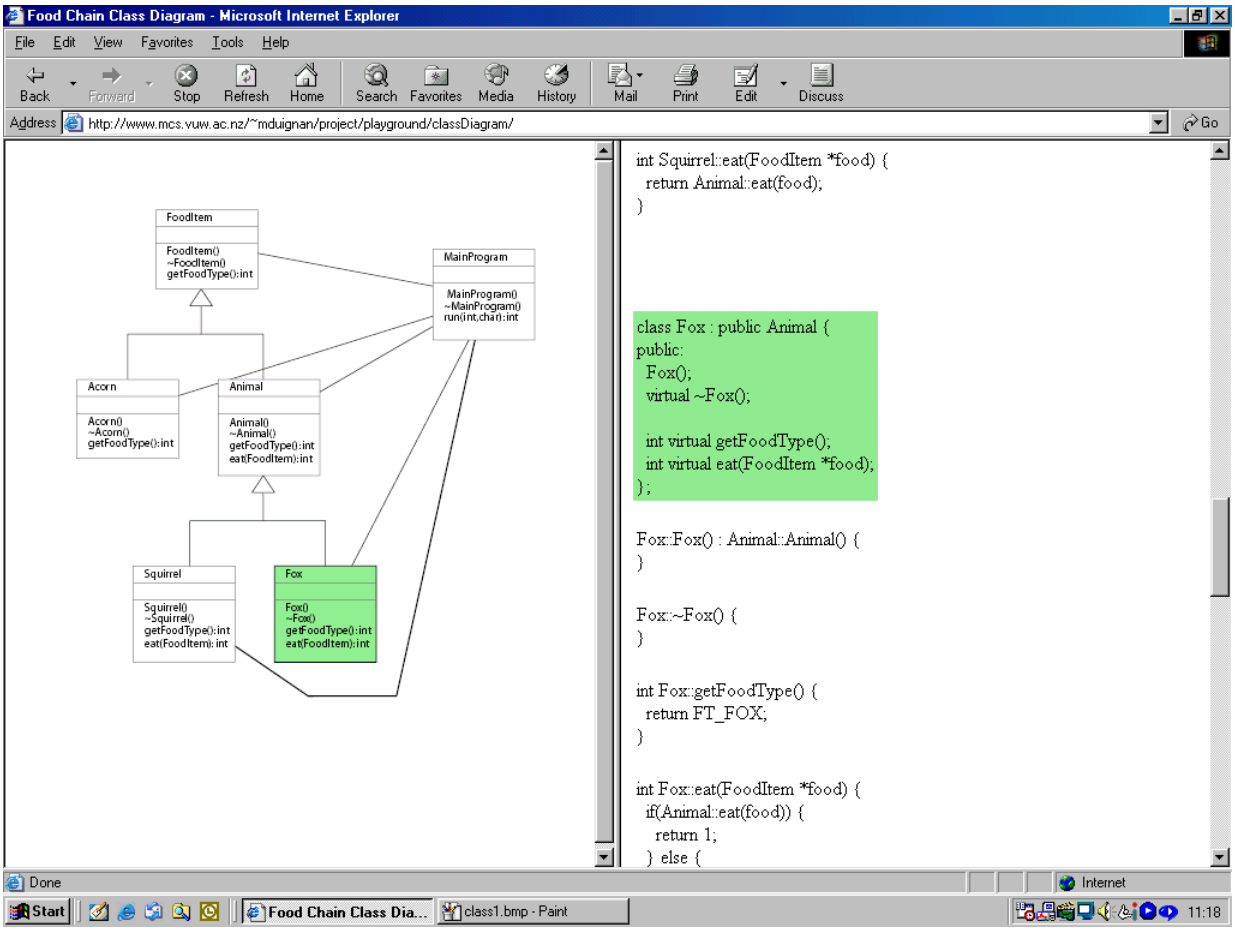


Figure 4.9: A UML class diagram I implemented in SVG which displays and highlights associated source code.

On clicking on a hyper-link from a web page to the visualisation, the user is presented with a two-sided display. On the left is a class diagram which depicts a simple program. On the right is an HTML document containing the source code for the program. As the user moves the mouse-over the various objects in the class diagram they are highlighted. This also triggers the source code on the right hand side to move to the correct position in the code and highlight the appropriate lines. The highlighting serves to show the linkage between the currently viewed source code and the associated object in the diagram. The object's methods can also be activated, which again highlights the appropriate point in the code — although this time with a different colour.

This was implemented by reusing the simple class diagram created in Adobe Illustrator. This SVG was embedded in an HTML page, which was then put into the left-hand frame of an HTML frameset. A second HTML page was put into the right-hand frameset. This page contained the source code of the program modeled in the SVG diagram. Appropriate scripting events were added by hand to elements in the SVG document that would be rendered as object boxes and method names. Strangely, Adobe Illustrator had represented the boxes as path elements, which is not an optimal implementation. This also made it difficult to find the appropriate element because instead of being a `rect` (for rectangle) element it was an obscure list of points in a `path` element. While in some cases editing the SVG by hand could be avoided by using Adobe Illustrator's script insertion mechanism, some SVG applications would be too complex to use this simple tool. Actually inserting the script was as simple as adding a scripting event attribute to the appropriate element. The event attribute for the Acorn class box was written as:

```
onmouseover="showLocal(evt); showCode(‘class Acorn’);"
```

This calls two ECMAScript functions. The first function is called `showLocal` and is located in the SVG inside a script element. It highlights the appropriate class box. This was done by changing its colour attribute via the DOM. The second function is called `showCode`. It is located in the SVG's HTML page. This function in turn calls another function located in the program code HTML page. This final function highlights and jumps to the appropriate section of the program's code.

Collapsible sequence diagram

The second interactive extension to UML developed for this evaluation was a collapsible sequence diagram. This allows the user to unfold the nested messages as they desire to see them. This could be particularly useful for users wanting to discover how code works if they were hoping to reuse it. They would avoid being overwhelmed by information, and would be in control of their own learning. Because this example was written by hand, fleshing out all of the details of a sequence diagram would have taken an immense amount of time. Construction of a complete example was left as an exercise for the automatic generation discussed in section 4.3.2. Instead, the actual construction focused on the design of the

structure, and the implementation of the scripting to make the diagram dynamic. Figure 4.10 shows a test for the design in a fully collapsed state. To uncover a message, the user can click on the square at the top of the method-call box. The message is then expanded, showing the next level of nested message, each ready to be unfolded in turn. Figures 4.11 and 4.12 show the diagram in successive states of unfolding.

Figure 4.10: The basis for a collapsible SVG Sequence diagram fully collapsed.

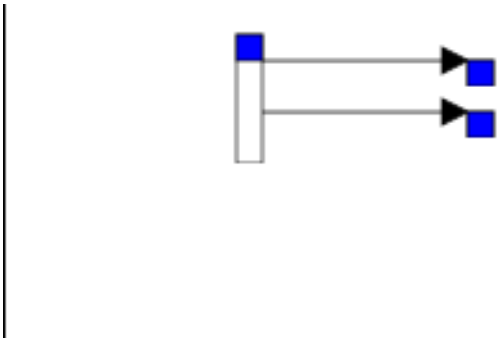


Figure 4.11: The basis for a collapsible SVG Sequence diagram partially collapsed.

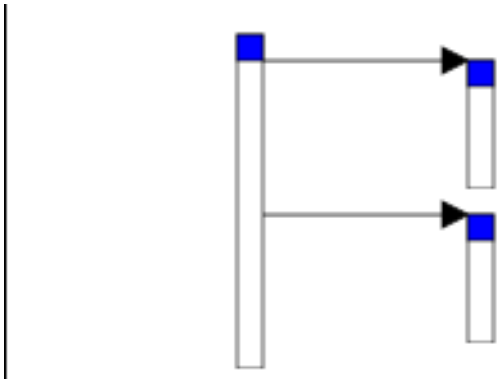


Figure 4.12: The basis for a collapsible SVG collapsible sequence diagram fully unfolded.

This was attempted in a number of different forms before a working design was found. Having a good structure was necessary to make the task of manipulating the SVG via script possible. The final design was based on embedding `svg` elements recursively inside one another. An `svg` element is the element normally used as the root of an SVG document, but it can be included recursively inside other `svg` elements. Each method-call is represented as one of these `svg` elements. These `svg` elements each contain two `animate` elements. One controls an animation that moves the `svg` up or down. The other controls the element's height. These

animate elements can then be used to move a method-call up or down as needed, as well as animating the method-call to collapse by decreasing the height to almost zero. The `svg` elements also contain two `rect` (rectangle) elements to draw the method activation box, as well as the little coloured square to be clicked on. Each `svg` element also includes a `use` element to include an appropriately positioned method-call arrow, and a text element for the method name. Finally, each `svg` element can contain any number of sub-svg elements. These sub-svg elements contain the details for method-calls called from this method-call. The code for a single method-call looks something like figure 4.13.

To implement the interaction, one hundred lines of ECMAScript code implement recursive algorithms for collapsing and expanding various sub-trees of the diagram. The general direction for the implementation was to have a complete representation of the diagram in the SVG and have the ECMAScript modify it as requested by the user. However, this method was pushed to its limits in this example. As SVG only provides mechanisms for storing the state of the displayed graphics, it is difficult to manage the semantic content of the graphic during interaction. To create anything more dynamic or complex would require using scripting to record the underlying state of the diagram, and using SVG as a raw medium. This is what was done in Adobe's Chemical Markup Language example. The reason for the need to move to a "script-centric" approach is that SVG only allows you to describe the current state of the diagram. Other information is needed in interactive examples, such as the collapsible sequence diagram. For example, when a method-call is collapsed, the uncollapsed height needs to be stored somewhere.

4.2.3 Statistical visualisation

Because of a lack of ready access to a large pool of program statistics for use in a diagram, I took a more abstract approach in testing SVG's capability for displaying statistical type visualisations. The SVG bar graph shown in figure 4.1 demonstrates SVGs capabilities for displaying basic aggregated data. For this reason I created a message mural type visualisation (see figure 3.1) to test SVG's capability for displaying raw data in a visual form. A simple algorithm was used to create an arbitrarily long mural which showed randomly generated data. If real data was used, the x-axis would encode the passing of time, while the y-axis would encode particular objects. Messages would be represented by the vertical lines. Colour could be used for some attribute of the message. Figures 4.14 and 4.15 show an example simulating 5000 messages.

4.3 VARE integration

The next section of the exploration focuses on the need for SVG to integrate into the VARE architecture. However, the benefits of this are not limited to a VARE-centric evaluation. The important learning here is in the discoveries found from generating SVG dynamically. Much

```
<svg height="70" width="2000" x="0" y="50">
  <animate attributeName="height"
           attributeType="XML"
           begin="indefinite"
           dur="1s"
           fill="freeze"
           from="0"
           to="70"/>
  <animate attributeName="y"
           attributeType="XML"
           begin="indefinite"
           dur="1s"
           fill="freeze"
           from="0" to="50"/>
  <rect height="40"
        style="stroke:black; fill:white"
        width="10"
        x="865"
        y="25">
    <animate attributeName="height"
             attributeType="XML"
             begin="indefinite"
             dur="1s"
             fill="freeze"
             from="0"
             to="40"/>
  </rect>
  <rect height="10"
        onclick="collapse(evt)"
        style="fill:blue"
        width="10"
        x="865"
        y="25"/>
  <use height="35"
       width="30"
       x="875" xlink:href="#LBmethodCallArrow"
       y="20"/>
  <text style="stroke:black"
        x="890"
        y="15">FoodItem(...)</text>


  ... Some number of sub-svg elements here
  <svg...
  ...
  </svg>

</svg>
```

Figure 4.13: An example of the SVG code representing a single method-call in the collapsible sequence diagram.

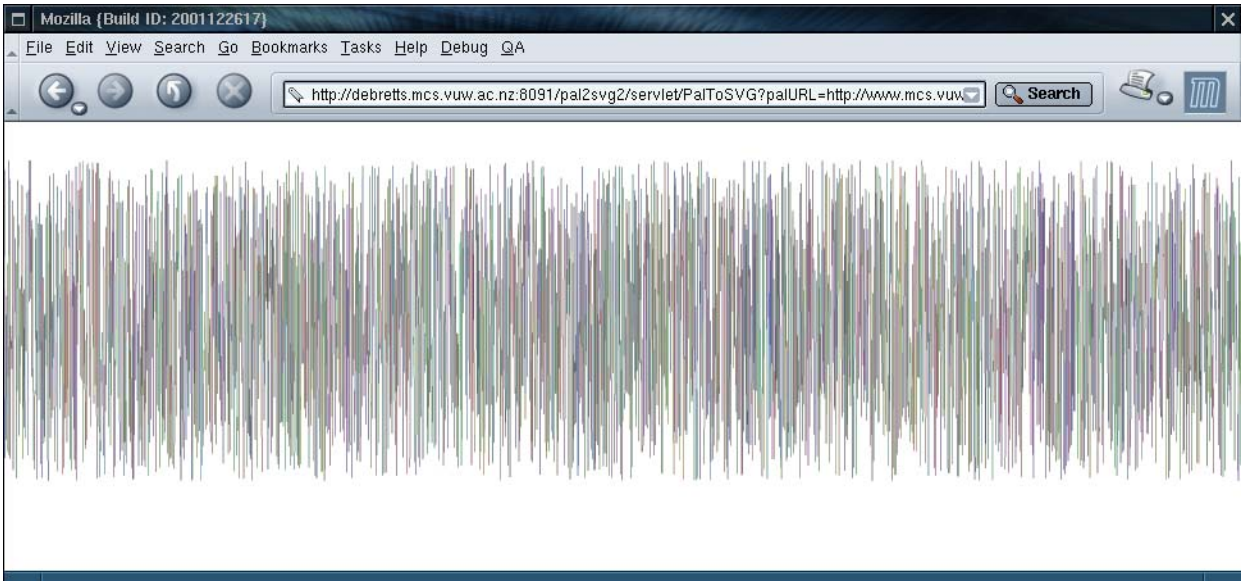


Figure 4.14: An SVG visualisation of random data mimicking Jerding and Stasko's message mural[26].

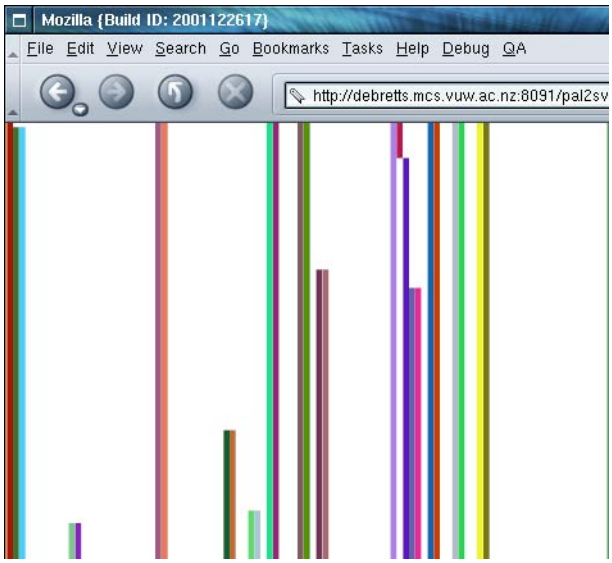


Figure 4.15: An SVG visualisation of random data mimicking Jerding and Stasko's message mural[26] under magnification.

of the field of software visualisation is involved with automatic generation of visualisations from an executing program, and so dynamic creation is an important area to investigate.

4.3.1 AT — The Process Abstraction Tool

Michael McGavin recently developed a tool to fill the role of an engine in the VARE architecture[38]. This engine was called AT (for Abstraction Tool) and was designed to utilise the GNU C++ program debugger to run "test drives" on Unix programs written in C++. AT loads the executable code inside the debugger and records the internal behaviour of the program as it runs. The process can be controlled remotely using SOAP, and AT produces test drive reports in PAL. A simple prototype web/internet interface was built by our research group for this, which can be seen in figure 4.16.

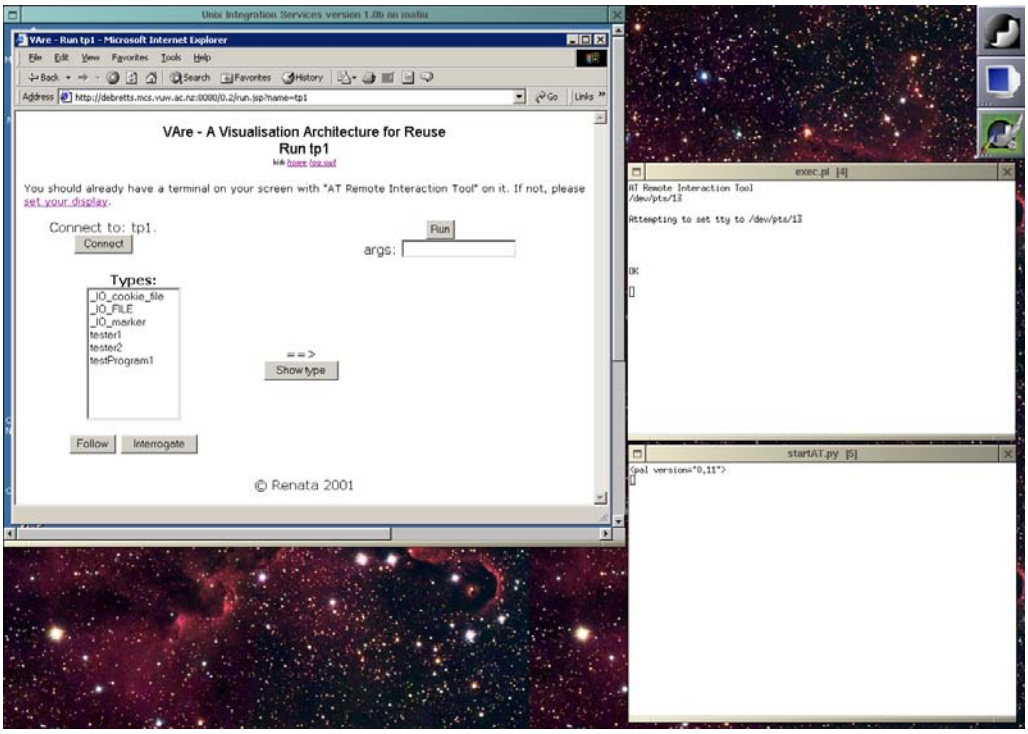


Figure 4.16: Test driving a C++ program via AT.

The web page on the left of the figure allows the user to pick a component or a complete program to test drive. The programs input/output is presented in the top right terminal, while the bottom right terminal shows the PAL output generated by AT.

4.3.2 Building a transformer

With a working engine and simple interface in action, building a transformer is the next logical step. This aids both the immediate development of VARE, as well as testing the dynamic creation of SVG. However, because the primary focus is evaluating SVG, as well as the still emerging nature of the VARE architecture, the transformer that I developed is not

controlled by SOAP. However, adding in a SOAP control layer would not be difficult. The transformer I developed and implemented is called Blur[2].

Java servlets

A convenient vehicle for delivering dynamically created web content is Java Servlets. Java Servlets integrate with web servers to provide dynamic content, but are platform and server independent[41]. They are written and compiled in the Java programming language, and simply need to provide a certain interface for a compliant web server to load and run them when requested from the web. Their output is then streamed through the web server back to the client web browser. I chose to use Java Servlets in the implementation of my transformer.

Transformer design

I created the Java Servlet PAL to SVG transformer in a typical modular design. A PalToSVG object handles Transformation requests from the web server. The PalToSVG object is passed a visualisation type to create, as well as a reference to a web URL which points to a PAL document. It then creates a PalParser object to turn the referenced PAL into a convenient program representation in memory. The PalParser utilises the Java XML parser technology "Xerces"[21]. Finally, the PalToSVG object selects an appropriate SVG software visualisation construction object to create the SVG from the program representation and stream it back to the user's web browser for display. Both the PalParser, and the implementations of the visualisation creators use the DOM interface implemented by Xerces to manipulate the XML they deal with.

Transforming code

As an example, let us now look at how Blur converts a class instance described in a PAL document into a corresponding SVG image. To do this, the PalParser includes code to recognise PAL's `type` and `event` elements. In this case, the `type` element we are interested in specifies the details of a class, including its name, methods, and inheritance structure. The element describing a class modelling a Squirrel could look something like the following:

```
<type name="Squirrel" typeid="t6">
  <context context name="sourcefile" contextvalue="tp7.cc"/>

  <classdata>

    ...the class data goes here...
```

[2]Blur was a character in the cartoon series "Transformers".

```
  </classdata>
</type>
```

The event element we are interested in describes the creation of a new class instance of a particular type (of class). The following PAL shows an example of an element which describes a new Squirrel object being created.

```
<event eventid="234">
  <newclassinstance classinstanceid="dcl232" typeidref="t6"/>
</event>
```

You can see here how the Squirrel type's `typeid` attribute value from the first code snippet matches the `newclassinstance` element's `typeidref` in the second code snippet. In this case the value is `t6`. This allows the PalParser to identify the class information for the new class instance.

When the PalParser encounters a type or event element in the PAL, it calls an appropriate helper parse method to add it to the developing program representation. For example, the PalParser's parseEvent method checks for the existence of a newclassinstance element inside the event element. If it finds it, it uses the DOM to gain access to the attributes in the following manner:

```
classInstanceId = element.getAttribute("classinstanceid");

typeIdRef = element.getAttribute("typeidref");
```

The values of the Java variables `classInstanceId` and `typeIdRef` will now contain the values `dcl232` and `t6` respectively. These can then be added to the in-memory program representation. Of course, building the program representation also involves binding this new class instance to its class type, and making this easily accessible for later parts of the program.

When the visualisation creation object wants to add a class instance to a visualisation that it is building, it can access it from the in-memory program representation. It extracts the information from the representation and creates a new `Element` object to insert into the SVG via the DOM. The following is some example Java code which achieves this:

```
// Get a friendly representation of the class instance from
// our program representation

classInstance = programRepresentation.getClassInstanceById("dcl232")
```

```
// Create a rectangle to represent the object in the diagram

Element object = svgDocument.createElement("rect");

// Set its attributes

object.setAttribute("height", "20");
object.setAttribute("width", "150");
object.setAttribute("x", someCalculatedXValue);
object.setAttribute("y", someCalculatedYValue);

// Create a text element to display the object's name

Element objectName = svgDocument.createElement("text");

// Create the actual text.
// Find the class name from the classInstance

Node objectNameText = svgDocument.createTextNode(classInstance.getName());

//Add the text value to the text element

objectName.appendChild(objectNameText);


//Add everything to the SVG docuement

svgDocument.getDocumentElement().appendChild(object);
svgDocument.getDocumentElement().appendChild(objectName);
```

In this example, a graphic of the class instance has now been added to the SVG document from its specification in a PAL document. The last step is to serialize the svgDocument and send it back to the client over the web.

Transformer user interface

I built a simple web-based user interface for controlling Blur over the Internet. This can be seen in figure 4.17. The user types or pastes the web address of a PAL file for conversion. The desired visualisations are then specified by selecting the appropriate check boxes. Each visualisation has a check box, along with an example screen shot and brief description. The user then clicks the "Build Diagrams" button at the bottom of the interface. The web browser opens a new window for each visualisation to be generated using ECMAScript. Each window

contacts the web-server and asks for the appropriate visualisation, passing through the URL of the PAL which is to be converted. Once the visualisations have been loaded, the user can browse the multiple views of the program execution that is described in the PAL document they specified.

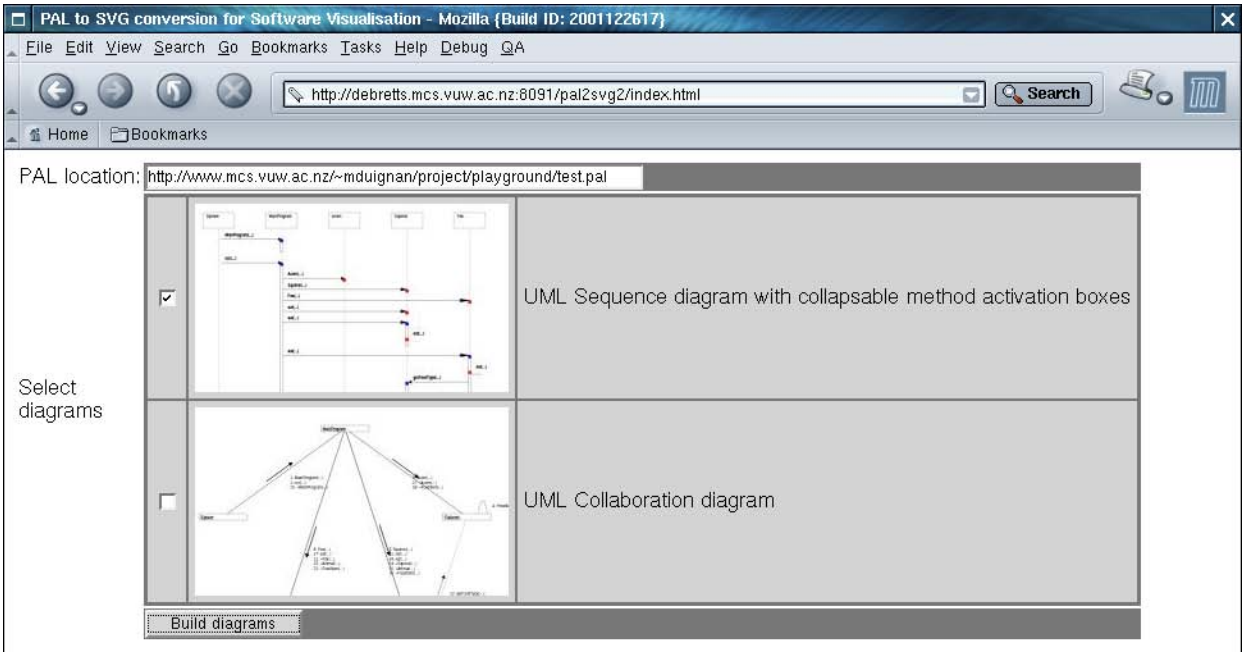


Figure 4.17: This simple web interface I designed allows a user to turn any valid PAL document into a number of software visualisations.

Dynamically created software visualisations

Four SVG visualisation creators were written for this system. They were built based upon the static visualisations from section 4.2.1, as well as a dynamic diagram from section 4.2.2. As discussed previously, each of these creators are used to turn arbitrary (but valid) PAL into visualisations. The following three visualisation creators were built initially, and a fourth was added later, and is discussed in chapter 6:

- A standard UML Collaboration Diagram creator. This builds a collaboration diagram by positioning the objects around a circle and uses trigonometry to calculate the coordinates. Trigonometry is also needed to calculate the rotation angle for the method call direction arrows. An example that was generated by blur can be seen in figure 4.18.
- A static UML sequence diagram creator.
- An interactive collapsible UML sequence diagram creator. This uses the same recursive `svg` element design as the simple example built by hand. Once the algorithm creates the structure of the diagram (by embedding the svg elements as required) it then uses a recursive algorithm to initiate all of the height and position variables. An example of the result can be seen in figures 4.19, 4.20 and 4.21.

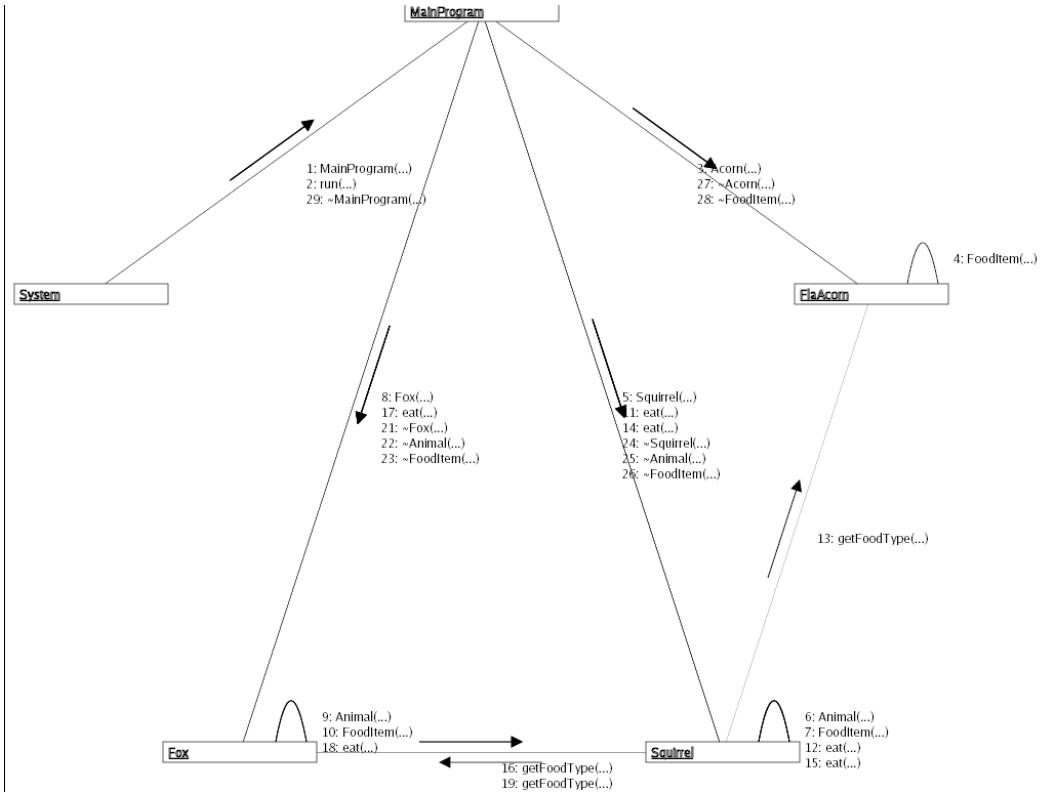


Figure 4.18: An SVG UML collaboration diagram generated by the transformer from PAL input.

Figure 4.19: A collapsible SVG sequence diagram fully collapsed.

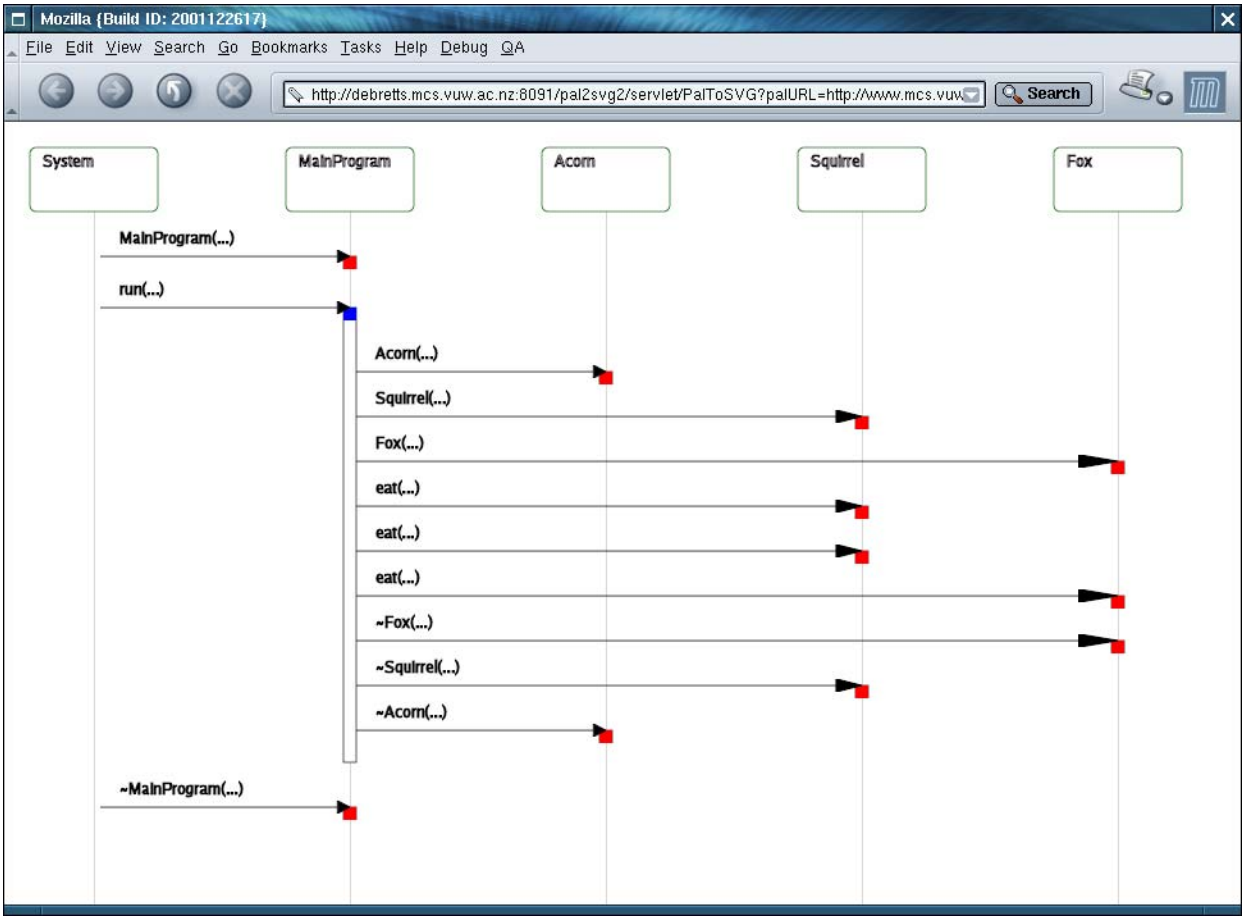


Figure 4.20: A collapsible SVG sequence diagram partially collapsed.

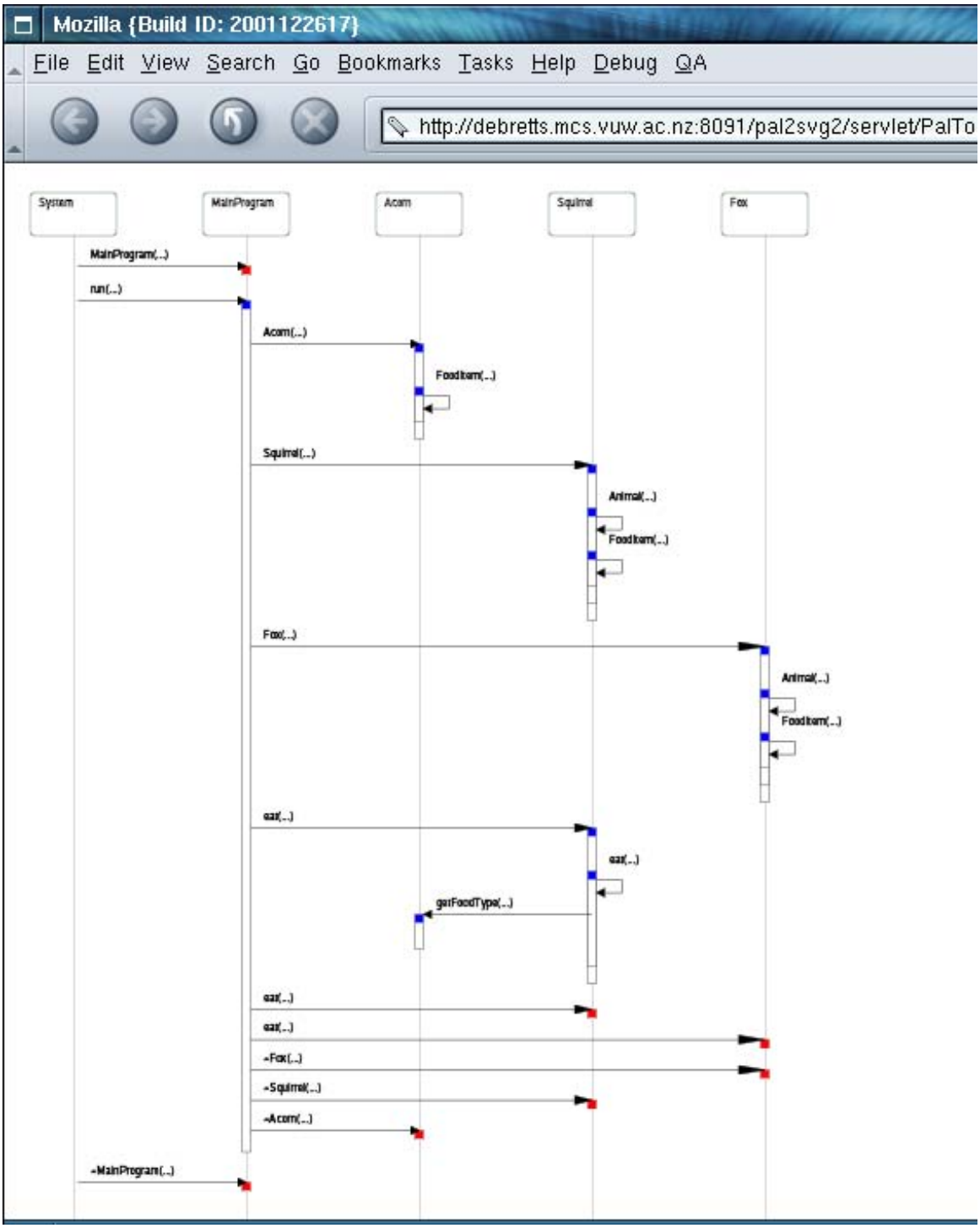


Figure 4.21: A collapsible SVG sequence diagram almost fully unfolded.

When a new visualisation type is needed, a new SVG creation object can simply be added to Blur.

VARE in action

Because a working PAL to SVG transformer has been built, it is now possible to follow through VARE's complete process of software visualisation generation. Since this work has been done primarily in order to evaluate SVG, there needs to be a certain amount of hand-holding. This could be addressed by the addition of a SOAP control layer, as well as the development of the central coordination component for VARE.

Using AT's web interface, a simple program was run which simulated various types of animals eating food. This produced a PAL output file, which was streamed to a file in a directory accessible via the web. The transformer's web interface was then used to enter the URL of the PAL file, as well as selecting both of the visualisation types for generation. Both visualisations opened in their new windows for viewing. This demonstrates that the process of using AT and GNU C++ program debugger to create PAL, and transforming these into SVG software visualisations is possible. A combined interface for controlling AT and the transformer could have been developed that would have automated the step from test driving to visualising. However, this is best left until further development of VARE's SOAP control system, and a carefully designed user interface has been developed. Figure 4.22 shows the interaction of AT and the SVG transformer. Comparing this with the VARE architecture shown in figure 2.4 reveals two things. Firstly, the lack of control lines between AT and the SVG transformer, as well as between the web interfaces, illustrates the current lack of a session manager and internal communication. Secondly, this current implementation has no repositories. It only has the single instances of AT and the SVG transformer.

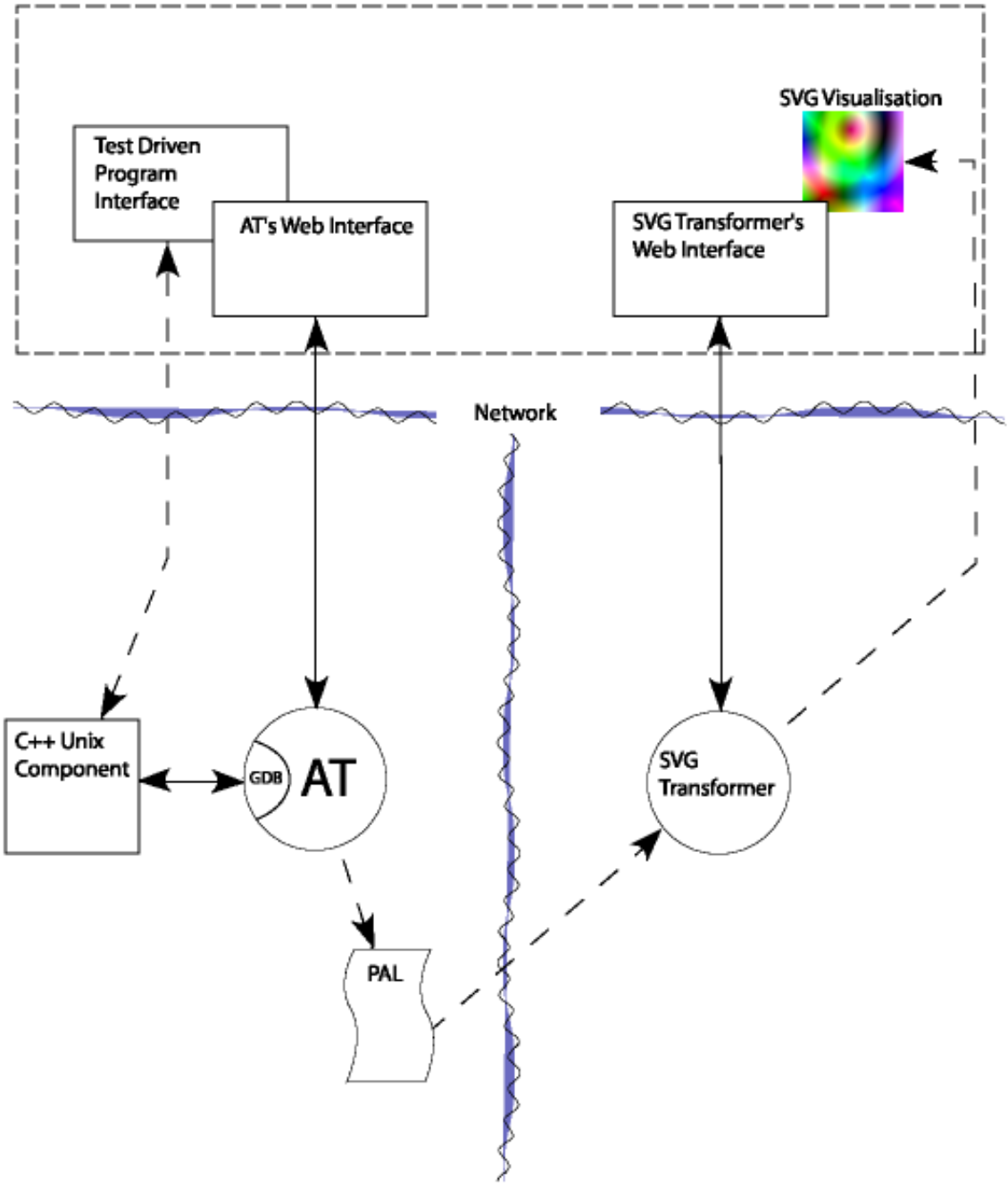


Figure 4.22: The simplified version of the VARE architecture implemented for this thesis.

Chapter 5

Evaluation

At this point, the experience gained from the work covered in chapter 4 is applied in evaluating SVG through the model developed in chapter 3. I present a brief summary of SVG's performance in each category of capabilities, and break these down in further detail. I then explore some of the interesting issues that emerge in more depth.

5.1 Basic information visualisation capabilities

5.1.1 Graphical capability

SVG performs well for most of the graphical capabilities. While it does not do particularly well with the spacial substrate, SVG is very able to support the remaining graphical demands. SVG can compensate for its poor performance with spacial substrate capabilities through the heavy use of scripting.

The spatial substrate

Dimensional support SVG provides a two dimensional coordinate system. There is no built-in support for representing additional dimensions. It is possible to implement three dimensions by calculating 3D to 2D coordinate conversions in scripts. However, trying to create true 3D content in SVG is largely prohibited by performance as 3D hardware cannot be utilised in this environment. SVG is not an ideal medium for 3D or higher dimensional use, due to this lack of built-in support.

Axis folding SVG provides no built-in support for axis folding. Every visual element must have an exact position specified with coordinates. Axis folding functionality could be implemented through heavy use of scripts.

Axis types SVG supports only straight axes. Other axes could be supported through scripting.

Axis distortions Arbitrary axis distortions cannot be established in native SVG. Again, this must be done via scripting. This can be achieved by ensuring that all use of coordinates on the distorted axis is mediated by specialised scripts. These scripts need to convert the virtual distorted axis values onto the real coordinates that are used in the native SVG. However, SVG does allow the specification of "transformations" on a coordinate system. These include moving, (universal) scaling, rotation, or skewing a coordinate space. This could be used to provide only simple axis distortion effects. These do not provide the programmer with a single global coordinate space with flexible distortions inside it. Rather, this cuts the space into a number of different coordinate spaces, each with their own different distortions.

Viewpoint control by system Pan and zoom changes can be specified declaratively in SVG. This is done by using animate elements which control transformations on the root SVG graphic. These animations can be staggered to create the moving viewpoint effect. Scripting can also control the "currentScale" and "currentTranslate" values to get the same effect.

Recursion SVG provides explicit support for subdivided space by allowing SVG elements to be embedded within each other recursively.

Marks and their properties

Size Size can be specified when shapes and paths are created. SVG has a competent scaling system.

Colour SVG has a sophisticated colour system which includes support for the International Colour Consortium's colour profile standard[8]. SVG allows a full range of colour specification as well as providing control via Cascading Style Sheets[57].

Orientation Orientation can be specified via SVG's competent translation capabilities.

Shape SVG allows the definition of paths which can include cubic and quadratic Bézier curves and elliptical arcs, as well as straight lines.

Points, lines and areas SVG allows for exact positioning of points in two dimensional space. Lines can be specified as described above. Areas can be defined by a path description and used for masking and controlling the area of graphic effects.

Filter Effects SVG allows a wide variety of filter effects.

Transparency SVG allows the specification of the transparency of any visible element.

Temporal encoding

Encoding time SVG has a timing model to allow changes to graphics over time. The animate, set, animateMotion, animateColour, and animateTransform elements all take

timing attributes so they can be cued at the appropriate points. Also, scripting allows programmatic control of the graphic.

Encoding identity The capability to specify animation declaratively (as opposed to manually setting values to fake animation) makes it easy to have visual elements move smoothly — and thus preserve identity.

Variation of retinal encodings Almost all graphical attributes of elements in an SVG graphic can be changed declaratively with the animate and set elements or via scripting.

5.1.2 Interaction

SVG is almost entirely dependent on scripting for interaction. Having said this, with scripting SVG is capable of a wide range of interaction types and is highly flexible.

Graphic malleability (after creation) Only with the addition of scripting can the structure and content of SVG graphics be changed at runtime. However, scripting can alter almost any aspect of the graphic arbitrarily at runtime. SVG's declarative modification elements (such as `set`) can only modify the attributes of the elements that are already in the document when it is first rendered.

Events The SVG specification covers a full range of pointer events, as well as focusin, focusout, and activate that could be driven from a keyboard. However, full keyboard support is not part of the standard. SVG 1.2 will probably contain support for navigating and triggering events in SVG documents with other input devices such as keyboards.

There are a series of event types cued when things happen to the SVG document, such as loading, resizing and closing. The standard also specifies DOM events which are triggered when the document is modified via the DOM.

Computation The only computation that can be done in native SVG is to allow alternative viewings to suit an SVG renderer's capabilities. A switch element allows conditional processing of parts of a SVG document based on these capabilities. Of course, the inclusion of scripting allows full computational abilities.

User notation SVG has no in built-in capability to allow users to annotate graphics. However, this can be provided with scripting, as in the example in figure 4.2

View refinement / navigation SVG renderers are expected to provide pan and zoom controls. The Adobe viewer does this via the context menu (right click) or by holding down the ALT or CTRL keys and left clicking.

Information hiding Graphical SVG elements have a visibility attribute that can be set to "hidden". They can be unhidden by setting the attribute to "visible".

Time control Time in an SVG graphic begins once the SVG is loaded. While animations can be restarted through native SVG event handling, accurate control of the "current time" can only be controlled via scripting. An SVG element has a setCurrentTime operation which can be passed the new "current time" by using the DOM.

5.1.3 Performance

SVG seems to have acceptable performance for small to medium sized graphics. Scripting and interaction seem to impose a large performance overhead, and animating text seems to be particularly slow in the Adobe implementation.

Scalability Increasing the complexity of a graphic will always have an impact on performance, but there is no indication that there are any inherent scalability problems in the graphic's specification of the SVG standard. On the other hand, SVG might have a weak point as scripting may begin to show serious scalability problems. This is because it is interpreted at runtime which imposes an additional computational overhead.

Current implementations The only fully featured implementation is the Adobe SVG viewer. It has now reached version 3 and its performance has steadily improved with each release. The viewer's current performance seems comparable to Macromedia Flash. While no solid empirical tests were conducted, very large versions of the message mural inspired example shown in figure 4.14 were tested. On a Pentium 4 1700MHz, 512 Megabytes of RAM running Windows 98 with the Adobe web plug-in, it performed fairly well. The test was fully scrollable and zoomable with fair responsiveness when there were up to 20000 line elements. Once the test was pushed up to 32000 line elements it became unusable, functioning with large delays. While drawing only straight lines is far less complex than curved shapes or text, this shows that even the current viewer is capable of displaying fairly complex diagrams. Another static example is the large class diagram from the Apache Batik project shown in figure 4.5. Even though it is very large, this graphic could be panned with ease. Zooming in and out worked, but with some delay. While Adobe's plug-in performed quite well with these static graphics, with interaction and animation examples it did seem to struggle a little. Even the fairly trivial version of the collapsible sequence diagram shown in figure 4.21 did not animate smoothly once the text for method names was added. Animation of text elements seems to be a weak point.

5.2 Software visualisation-specific and higher-level capabilities

5.2.1 Integration

SVG has a huge strength in integration. Due to its XML basis, as well as it being developed for the web, it has unique advantages over other graphics formats.

Creation mechanism Since SVG is based on the plain text XML standard, there are a wide range of options for creating graphics with it. A SVG document can be hand-crafted in a text editor. A vector graphics drawing program such as Adobe Illustrator[53] can be used. XSLT can be used to create an SVG document from an XML data source (but there is a limit to how this can be done usefully which is discussed in section 5.3.1). Standard XML tools can be used to create the diagram programmatically.

Deployment SVG already has the MIME[50] type of image/svg+xml[61]. MIME types are a standardised way of declaring and transmitting non ASCII file types over the standard ASCII protocols of the internet[1]. By using this MIME type, SVG can be transferred by existing web servers over the internet as requested. Web pages can embed or link to SVG content for easy access. SVG can also be sent via email, or distributed in any other form. Users will currently need an SVG viewer installed on their machine, but these are free and easily available. In the future, SVG will likely move to being included inline in XHTML, and web browsers will render SVG natively. Indeed, Mozilla already supports this[44]. However, SVG can not easily support live program software visualisation viewing. This is discussed further in section 5.3.2.

Linkages with other technologies at display time SVG is very strongly integrated with scripting. The scripting language to be used is not dictated by the SVG specification, and it can therefore integrate with any scripting language and implementation needed. The most popular choice is a variation on javascript or ECMAScript. However, once the scripting language is specified, the SVG document can contain script elements as well as script event attributes in the chosen language. SVG integrates well with the web. Any visual element can be a hyper-link to any other web resource.

View coordination with other visual media SVG is dependent on scripting to coordinate multiple views across media. The only limitation is in the scripting environment used in an implementation. In principle this is not limited by SVG. In practice, multiple SVG views can be coordinated within a web browser's scripting environment. It is important to note that this functionality is outside of the SVG standard, and therefore cannot be guaranteed to be compatible across different implementations. Testing with interaction with other media (e.g. Java) was not conducted.

[1] MIME stands for Multipurpose Internet Mail Extensions, which betrays its origins as a standard for email attachment types.

5.2.2 Higher-level capabilities

SVG is disappointingly lacking in higher-level capabilities. After becoming accustomed to programming in well developed user interface frameworks, moving to SVG is a shock. SVG is quite low-level. This is fine for working with simple graphics, but becomes troublesome when applied to more complex applications. Again, scripting is necessary for any hope at achieving these capabilities. The only exception to this is in built-in graphical capabilities where SVG performs rather well.

Higher order graphics SVG provides support for a number of basic shapes including rectangles, circles, ellipses, lines, poly-lines, and polygons. At specification they can all be given appropriate attributes such as position, width and height. The lines and curves supported by SVG can be used to build non-standard shapes. These non-standard shapes can be specified as "symbols" and used repeatedly in a graphic. Sophisticated layering and filtering (as already mentioned) are part of the standard. Gradients and patterns (textures) are supported.

Basic text is well supported in SVG with provision for multiple languages and fonts. However, SVG 1.0 does not support automatic word-wrap which complicates the use of text in SVG hugely, but SVG 1.2 will probably address this. Also, moving text seems to be a point of performance slowdown.

Data/display independence There is little room in SVG to store data independently from display. An SVG document is only designed to describe a graphic, not to store other forms of information. There is a meta-data element that can be included to describe any SVG element, although this is meant to be used for descriptive information rather than raw data. However, scripting can be used to store the data while it creates the content of the SVG on the fly.

The order and groupings of graphical elements in an SVG document determine the layer ordering for the display. This makes it impossible to group and order elements to capture semantic meaning when this conflicts with the elements layer in the final graphic. SVG 1.2 may provide a drawing order attribute to deal with this.

Referencable Entities/Objects Every graphical element in an SVG graphic can be referenced via the DOM or XML ID attributes.

Layout constraints SVG provides no support for layout constraints.

Structure Elements can be structured with group (g) elements. However, these groups determine layering order (what elements lie on top of which others) and so are limited in their use for grouping elements logically.

5.2.3 Support for current software visualisations

SVG is able to display all three of the common forms of software visualisation. However, it does not make it as easy as would be ideal.

Node-link diagrams SVG can cope well with drawing node-link diagrams. However, the lack of layout constraints makes SVG a much less convenient medium to work with than it could be. Coordinate points must be specified absolutely, and cannot be given in relation to other objects.

Statistical diagrams The performance of the message mural inspired test has already been described in the performance section (5.1.3)of this evaluation. This indicated that SVG was quite capable of displaying non-trivially sized raw data displays. It would not perform well in more interactive and dynamic raw data displays. Aggregated data is displayed with ease.

Source-code-related diagrams While SVG can display text, most implementations would probably have trouble rendering the huge quantities of text that make up the source code of large programs. However, SVG's integration with HTML more than makes up for this in many cases due to HTML browsers' relatively good performance with text. For many source-code-related diagrams, it would be possible to have the graphical segments in SVG while the source code segments are in HTML. The SVG would simply have to be embedded in the appropriate places in the HTML, with scripting used to let the HTML and SVG work together. However, some visualisations could be conceived that could not be divided like this, and they would push SVG renderers to their limits.

5.3 Points of interest

Now that I have briefly covered how SVG performed in each of our model's capabilities, a more detailed discussion of the interesting points that were raised can be undertaken.

5.3.1 SVG creation

SVG has a number of interesting options available for programmatic generation and creation.

XSLT

XSLT is a declarative style sheet language for styling XML documents. It is often used for XML data conversion, moving data from one XML document type to another. As such, it would seem to be a logical choice for creating SVG from PAL. However, XSLT's primary purpose is as a style-sheet language, and its declarative nature, make it awkward for the types of processing needed for visualisation creation. While XSLT is good for simply changing the

form of XML data, it would be difficult to use XSLT to implement some of the complex layout algorithms needed to convert raw data into visualisations. However, XSLT can be used to generate script content for an SVG document which will be run automatically when the SVG is loaded. This SVG script content can then build and control the graphical elements required in the visualisation. An example of this could work as follows:

XSLT could be used to parse PAL, and for every type definition it finds it could write a insertType(...) function in the SVG script content. The XSLT parsing of the PAL could also recognise elements in the PAL that represent events and write the appropriate insertEvent(...) functions in the SVG script content. When the SVG — which is empty apart from the XSLT created script content — gets to the client for display, the scripts are automatically run, building the visualisation on the fly. This example is inspired by Adobe's Chemical Markup Language demo shown in figure 4.3.

However, what this amounts to is developing visualisation creation transformations in two contrasting environments. One, (server based) uses XSLT. The other (client based) uses script. The end result is a transformation system of far more complexity then is needed. So while XSLT may have a place in some SVG software visualisation generation systems, it is unlikely to offer a complete solution.

Server side generation

The other alternative is to complete the generation of the visualisation entirely on the server side. This is more in line with the VARE architecture, as well as being the simpler and cleaner approach. As explained in chapter 4, a Java transformer was implemented that could be used as a Servlet. However, the programming language used could have easily been Perl, C++ or Python. The main issues here were that the environment is controlled (not on an undetermined client system with an unpredictable setup), and that a quality XML toolkit was available[21]. The presence of an XML toolkit simplifies the parsing of PAL, as well as the creation of the SVG.

The original hope was that the PAL input could be parsed by the standard XML parser and visualisations could be created by accessing the PAL via the DOM. In practice, although accessing PAL via the DOM is higher-level than direct file input/output, it is still too low level to be a practical source for creating a visualisation. To be worrying about "what the third child of the fourth element of the next event element" is when creating visualisations is very awkward. To further complicate the matter, PAL has been designed in such a way, that discovering all the information about a particular event or type often requires following multiple "ID" references to get all the necessary information. This is the reason that the transformer creates an in-memory representation of the program to be visualised. It is then easy to create the SVG from this more convenient representation.

While the DOM was chosen because it was hoped it could fill the role of the program representation, its failure to live up to this means that SAX could be equally useful.

Writing SVG by hand

Creating a complete software visualisation by hand is possible, but painful. Editing SVG at the source code level is more suited for either building basic components for use in programmatic creation, or for fine-tuning SVG output from a drawing or conversion tool.

Drawing tools

Using Adobe Illustrator[53] proved to be an easy way to create one-off SVG diagrams. However, the code it produces tended not to be as conceptually tidy as what would be produced by a human. For this reason, this is not an ideal route if lots of work needs to be done to the SVG after export (like the addition of complex scripting).

UML CASE tool file conversion

Finding a way to create SVG from a CASE tool designed for creation of software visualisations is an attractive option for some situations. However, you are limited to creating only the well known, supported visualisation types. This would be too limiting for some applications, but useful in others.

5.3.2 Streaming SVG

SAX's simple event triggering design makes it ideal for streaming conversion of XML content. A form of streaming could be used to animate a live visualisation of a program execution on a server. However, doing this with SVG is problematic. While some implementations of SVG renderers will support a crude form of streaming in that they display the SVG as it becomes available, this is too limiting for running even moderately interesting animations. This is because a stream of SVG can only add to the SVG which has already been rendered at the client — not manipulate or change what is already there. To do serious live program visualisation with SVG over the Internet, another agent would need to be present on the client. This agent would read an event stream and modify the SVG via the DOM. This could be done via scripting, but Java would be a more attractive option. If scripting in SVG was chosen, a communication path would need to be established with a server for updates. This is not part of the SVG standard, but is offered in some implementations as an extension. Both Adobe's plug-in, and the Apache Java SVG viewer provide a getURL function in their scripting environments which would allow this. Streaming will probably be directly supported by SVG 1.2.

5.3.3 The importance of scripting

A theme emerging from the evaluation so far has been SVG's reliance on scripting to support many of the capabilities which would prove useful for software visualisation. The "dirty"

nature of scripting, as implemented in practice, combined with the fact that the SVG specification defines no standard scripting language, means that the developer must be very careful. While doing simple scripting work in a homogenous environment (e.g. where all web-browsers are the same) is acceptable, the implementation of the complex script capabilities required for SVG software visualisation could cause problems. This is caused by the fact that scripting environments are not truly standardised across various platforms. Because architectures such as VARE cannot afford to be tied to a single platform, presumptions about client implementations must be kept to a minimum. However, if SVG with scripting was used, the following could be achieved:

XSLT and client side generation

The details of this have already been discussed in section 5.3.1.

3D with shadows

Adobe's Chemical Markup Language example (shown in figure 4.3) shows a 3D visualisation with shadow effects on SVG's 2D plane. This shows that SVG can be used simply as a raw display medium while graphics libraries are implemented on top in a scripting language. However, traveling too far down this road would certainly raise serious performance problems. At some point it would need to be asked, why not use some other medium that supports these capabilities natively without the performance hit of interpreted scripting, and lack of hardware acceleration? Other options could include 3D markup languages such as VRML[10], the emerging X3D[9], or alternatively Java with Java3D.

Improved node-link diagrams

While SVG is more than capable of displaying node-link diagrams, the underlying representations that are possible are far from ideal. Unfortunately, for reasons discussed shortly in section 5.3.5, SVG does not support a line to be specified as linking two other graphic entities. This becomes a concern when you need to move one or more of these entities. For example, a user might want to modify the layout for clarity. If the graphical entity is moved, any lines that linked to it will become orphans. The only way to address this is to track the "nodes" and "links" in scripts, and demote the SVG further towards the the role of a raw medium.

Data display independence

Again, data display independence is reliant on scripting doing the work behind the scenes, while the SVG takes a passive role. In this case, creating a visualisation would involve determining the logical elements which will be present in the diagram. The SVG should arrive at the client in an empty or basic initial state except for the scripts which would take control when the SVG is viewed. Any interaction would need to be fully mediated

by the scripts, which would update their state, and then feed this through to the display (implemented by the SVG). In this case, you could begin to look at XSLT for translation. However, it would probably be better to use something more sophisticated on the server to create the entire initial state of the visualisation.

Logical structure

Logical structure (as opposed to the largely graphically determined structure imposed by SVG) would also be facilitated by the data display independence just outlined.

5.3.4 Creating objects from symbols

SVG seems to have a strange weakness in how symbols are created and used. To make SVG easier to use (and more efficient for downloading) recurring groups of SVG elements can be defined as a symbol. These symbols can then be used as many times as needed in a graphic. This is done by specifying a `use` tag, and giving coordinates for its location in its attributes. Unfortunately, there seems to be no easy way to specify other customisations at creation time. For example, in creating the rectangles at the top of a sequence diagram, all that needs to be customised is the value of the text element which displays the object's name. However, this cannot be done through the declarative means of the SVG's `use` element.

5.3.5 Layout constraints

A weakness that seems to be a result of SVG's philosophy is the lack of any support for layout constraints. SVG's philosophy seems to be that all graphical elements should be placed in exact coordinates. In other words, SVG makes no effort to help in the intelligent placement of objects. The programmer must do all the work. For example:

- There is no way to automate line wrapping for text which runs off the screen or outside an area.
- There is no way to specify an element's coordinate values as being an expression to be calculated.
- There is no way to specify behaviour rules for specific elements (e.g. different zoom factors when a zoom is triggered).

These limits can be overcome through diligent script coding, but it seems a shame that they were not included in the base SVG specification.

One interesting attempt to extend SVG to allow layout constraints is pSVG[32]. pSVG stands for "parametric SVG", and allows developers to embed variable parameters and expressions in SVG content. Figure 5.1 shows some example pSVG and a description of how it works.

```
<rect x="{$x-2}" y="{$y-2}" width="4" height="4"/>
```

The pSVG processor will first convert all property name references to their values. Let's say that our context-object has these values: x: 10, y: 20 . After phase one is complete, our pSVG string will look like this:

```
<rect x="{10-2}" y="{20-2}" width="4" height="4"/>
```

Phase two will process each delimited JavaScript section. When you eval() a statement like, "10 - 2", eval() will return 8. So, after each section of delimited JavaScript is eval()'ed, we end up with the following string:

```
<rect x="8" y="18" width="4" height="4"/>
```

Figure 5.1: Example code and description of Parametric SVG from KevLinDev.

While pSVG essentially provides all that is needed for layout constraints, it is unfortunate that it is not included in the SVG standard directly. Because it is in fact just a service provided by pSVG ECMAScript objects, it is impossible to include pSVG directly in the body of the SVG document. This means that all of the parameterised SVG content must be added to the document through ECMAScript calls. It would be great to see some of these ideas incorporated into SVG as this would simplify practical use of SVG greatly.

5.4 Alternatives to SVG

While this is primarily an evaluation of SVG as a medium for software visualisation, it is also prudent to at least mention some of the alternatives. However, this discussion will be limited to a brief comparison with SVG. A more complete exploration would require measuring each of the alternatives against the evaluation model in order to usefully develop the points of comparison.

5.4.1 Macromedia Flash

Flash provides very similar basic capabilities to SVG. In fact, there is a working (but still developing) conversion script from SVG to Flash's file format (SWF)[29]. However, Flash has a number of crucial differences. Firstly, while Flash has a vector based file format, it is a binary format. While this can result in better performance, it has a major drawback. Creating new authoring systems is made very difficult. For both hand-authoring and programmatic generation, Flash has only a few tools available [43][35][34]. Compared to the breadth of support for authoring SVG via general XML tools, Flash's options are limited. Additionally, because SVG is backed by standard XML, high quality authoring environments already have SVG support[53], and more will probably follow. While Flash's format is a published standard, it is not an open comunity standard which means that the real control

of the future of Flash lies with Macromedia. An interesting development in the latest version of Flash (Flash 5) is that it now has direct support for XML data communication[36]. What this means is that a Flash graphic being viewed on a client machine can now access XML data from the web as needed, as well as send XML to servers. This built-in functionality would make it easy to make a client Flash display talk in SOAP with the VARE server. This is a clear advantage over SVG. It could even go as far as to read PAL itself, and implement a transformer, even if this might not be the most elegant design. Importantly however, this does not overcome Flash's central weakness. The XML integration is only an addition to its capabilities, *not* its file format. This means that developers are still limited to using the small set of tools that are available for Flash authoring. Finally, one big advantage of Flash is that it has support for streaming "movies", which would be useful for live program visualisation.

5.4.2 VRML and X3D

Another option is to use VRML, or the still emerging X3D as media. Like SVG, these technologies are for the display of graphics over the web and are also based on markup languages. However, they would probably not be ideal for the two dimensional applications that dominate much of the software visualisation field. Further examination of these media would be necessary in order to provide further comment.

5.4.3 Java

Another option would be to utilise Java. Java Applets[39] allow programs written and compiled in Java to run in a web browser on (potentially) any hardware/software platform. This is a very interesting option, as it provides a sophisticated graphical user interface toolkit, as well as full network capabilities for intelligent communication with a server after deployment. This could include communicating in XML based languages (such as SOAP) to integrate with an architecture such as VARE. However, Java is clearly a heavy-weight solution, while SVG is the light-weight alternative. SVG allows the developer to deploy simple descriptions of graphical content, rather than executable programs. For small software visualisations, such as some UML diagrams, SVG would be a perfect choice. Larger, more dynamic visualisation would probably be better served by Java applets. Interestingly, SVG content can be included in Java applications through the Apache Batik libraries[20], so SVG visualisations could be displayed from within larger Java visualisations.

5.5 SVG's strengths

SVG has many good features and a few great advantages. SVG's biggest strengths are as follows:

- SVG was born for the web. SVG integrates well in existing web browsers through current plug-ins, and if the built-in browser support matures this will be a powerful

advantage. Being able to embed SVG natively in XHTML web pages will allow clean delivery of mixed HTML SVG content. This will be useful for source code based visualisations, as well as for giving the user a seamless code discovery experience. All of this can be delivered over the Internet to a wide variety of platforms.

- SVG provides an easy way to create high quality graphics. The built-in capabilities for textures, filters and effects provide a good set of tools to get the quality visual results required in software visualisation. Being vector-based makes creating animation and simple interactivity easy.
- Because SVG is XML based, SVG can already be created and manipulated by a huge variety of tools. Also, XML's popularity makes coding SVG easy to pick up for a large proportion of programmers.
- SVG's light weight nature makes it much easier and faster to deploy than more heavy-duty options.

5.6 SVG's weaknesses

Of course, SVG has a number of weaknesses. The most prominent are as follows:

- While SVG performs relatively well for small to medium-sized graphics, it will probably never be able to compete with dedicated and compiled programs when it comes to large visualisations with complex interfaces.
- SVG does not have any support for layout constraints. Therefore, programmers are forced to deal with writing code to manage this by hand in client side interpreted scripting languages.
- SVG is extremely dependent on client side interpreted script for supporting many common capabilities that software visualisations demand. This sort of scripting environment offers little standardisation, poor performance, and has a lack of easily available and mature development environments.
- This reliance on scripting for typical common requirements results in a situation where the optimal deployment is an SVG document containing only script. The script then does all the work, using the SVG graphics elements as only a type of low level graphics control mechanism. The fact that writing non-trivial SVG actually means creating ECMAScript "programs" is something that many programmers would be unhappy with.

5.7 Possible improvements for SVG

5.7.1 Layout constraints

Adding layout constraints, as proposed by Tirtowidjojo et al[56], could bring SVG to just a high enough level to avoid much of the dependence on scripting. Attempting to add layout constraints to SVG with ECMAScript, as is done with pSVG (section 5.3.5), is one approach. However, including layout constraints inline in SVG itself would simplify things immensely. Representing layout constraints in the SVG itself could not only be easier than writing scripts, but switching back and forwards between the world of scripts and the world of SVG markup would no longer be necessary. A possible suggestion for a layout constraint mechanism is to allow executable statements for numeric attribute values. This would result in being able to write SVG code like the following:

```
<line x1=SomeBox.width y1=(SomeBox.height + 10) ...
```

What this amounts to is taking a simplified and clearly standardised subset of script-like capability and cleanly integrating it into the SVG standard itself. This would allow more appropriate support for node-link diagrams, as the end of a line could be defined as being located wherever the center of a rectangle was. The line would then follow the rectangle wherever it was moved. With some more additions, this could also allow for automatic column wrapping text, and other high level capabilities. These changes would consist solely of additions to the SVG standard, which would not impact on the use of the current syntax. Performance should not be impacted too greatly, as otherwise all of this work would have to be done with scripting. Additionally, SVG renderers could be optimised for this functionality. This optimisation could not be done if non-standardised interpreted scripting is used, as it is at present.

5.7.2 Entity construction

Another possible improvement to SVG would be to allow symbols to be used in a more powerful manner. As it is currently, entities with similar but not identical properties cannot be abstracted out to a common symbol in the SVG document. For example, if a diagram contained a number of stick figures, each with either a smile, frown or grimace on its face, it is not possible to create a generalised stick figure symbol that allows the specification of different faces when used. SVG's `symbol` and `use` facility would be more useful if you could do things like the following:

```
<symbol id="stickFigure" ...
  <option id="smile" ...
    ... put smile shapes here
  </option>
```

```
  <option id="frown" ...
    ... put frown shapes here
  </option>
  <option id="grimace"
    ... put grimace shapes here
  </option>

  ... Put all of the common shapes
      i.e. the body here ...
</symbol>

...

<use xref="#stickFigure" option="smile" ...
```

5.7.3 Upcoming improvements

A discussion of possible improvements for SVG would not be complete without further examination of what is likely to be added to new versions of the standard. While the SVG 1.1[62] candidate recommendation will not contain any new features, the SVG 1.2[63] working draft looks a little more interesting. The following is a discussion of the new features being planned for SVG 1.2.

Text wrapping

SVG 1.2 will allow automatically calculated text wrapping in arbitrary shapes. This means that it will become trivial to define a shape in SVG, and then have text sit inside this shape, and line-wrap with the contours of the shape. This will be a huge advancement for the use of SVG in text-based graphics. To achieve a similar effect currently requires some fairly complex scripting. Integrating this into the SVG renderer itself should result in marked improvement, and simplicity of development.

XML integration

While SVG is XML itself, and can be included inside other XML documents, there are further improvements that can be made to integrate it with other XML standards. SVG 1.2 will aim to make a number of changes in this regard as follows:

- XForms

  XForms[66] is an XML application designed to replace current form implementations embedded in web-pages. XForms are designed to be embedded in other XML display

documents, such as SVG and XHTML[67]. XForms essentially provide all of the functionality for creating basic interactive user-interfaces. They provide various input fields, and the separation of presentation from content.

Once XForms can be integrated into SVG, it will greatly simplify the addition of standard user-interface components to SVG. This will reduce the dependence on complex ECMAScript as typical user-interface tasks will no-longer require it.

- XML Events

  The XML Event standard[68] provides a means of clearly specifying event listeners, while separating them from the actual XML content. Once this is integrated into SVG, developers will be able to specify event listeners separately from SVG content, improving modularity and maintainability.

- More SMIL Animation

  The Synchronized Multimedia Integration Language (SMIL)[65] is designed for the specification of interactive audiovisual presentations. SVG 1.0 currently supports SMIL for its animation facility. When more SMIL functionality is introduced in SVG 1.2, it will likely be possible to include video and audio content in SVG documents, as well as utilise more of SMIL's timing controls.

- Rendering Arbitrary XML

  As in the case with Blur, SVG is often being used as the display technology for XML data. Blur takes XML program descriptions (in PAL) and creates SVG visualisations from them. The SVG working group is currently looking at ways to support this use case, by incorporating translation features into the SVG standard.

  It would be interesting to see if there is any real benefit to this for the field of software visualisation, given the complex nature of the data being visualised, and the technical requirements of layout algorithms.

Printing

When SVG content is required to be printed, there are a number of issues that SVG could take into account, and some of these are being looked at for inclusion in SVG 1.2. While these new features are probably not so vital to software visualisation (as much of it is often based on interaction) there would be a number of static reference type visualisations which would benefit from these additions.

The likely changes include the ability to specify how SVG content should be broken across pages when printed, as well as specialised color support for the physical printing process.

Changes to the rendering model

- Alpha compositing

There are a number of changes to the compositing model proposed for SVG 1.2, but this is an area which is still under heavy discussion. These proposed changes will provide a more advanced system for the layering of graphics on-top of each other. The enhanced alpha compositing proposed for SVG 1.2 will allow the specification of how semi-transparent objects are blended together. SVG 1.0 and 1.1 do not provide such flexability, as they have only a single compositing scheme.

- Drawing order

  SVG 1.2 may allow the specification of drawing order as being separate from an elements position in the SVG document. This could aid developers in creating semantic groupings in their SVG documents, rather then be required to always group elements for graphical reasons. This could have a number of positive effects, including simplifying the management of an SVG document tree from ECMAScript, and aiding understanding of document structure.

Streaming

There are plans to introduce streaming into SVG 1.2. This would be vitally important for real-time program visualisations, as well as allowing time-based visualisations with a very long duration to begin displaying before the download is complete.

Painting

A number of changes are proposed in SVG 1.2 to enhance document structure with regard to color references, and text transformations.

Changes to the SVG DOM

There are a number of proposed changes for SVG 1.2 which improve the accessibility of various types of information via the SVG's document Object Model (DOM). These changes will not affect SVG's capability for software visualisation directly, but will ease some ECMAScript programming tasks.

Navigation

SVG 1.2 will likely enable input devices other than the mouse to navigate an SVG graphic. This could enable other interaction methods with the SVG document. However, the real impacts of this would probably be for the accessibility of SVG to those with disabilities.

Chapter 6

Building on SVG

It is clear that for SVG to become a useful medium for software visualisation, it needs to better support higher-level software visualisation capabilities. Because improvements to the SVG specification could be a long time coming, other possible remedies for its short comings should be considered. The absolute need for scripting in non-trivial visualisations means that a lot of time could be wasted in "reimplementing the wheel". If SVG is chosen as the primary visualisation medium for VARE — a role that it could surely fill — it would be very helpful to create a SVG software visualisation scripting library. This would greatly ease the job of visualisation development. Such a library can be referenced by a SVG script element and could include all of the functionality discussed in section 5.3.3.

6.1 Graphics APIs

Creating sophisticated software visualisations with SVG becomes a large ECMAScript programming task. However, the programmer must still be intricately involved with the details of the XML SVG content, setting attributes of graphical objects via the DOM. The result seems to be less then ideal. While many programmers are probably used to graphics programming using a specialised graphics application programming interface (API), SVG programming through ECMAScript provides a very different, and more complicated system. For example, the popular 2D and 3D graphics API OpenGL[46] provides function calls which allow the programmer to specify shapes as a series of vertices, by calling appropriate functions as follows:

```
// Draw a square (by specifying vertices of a polygon)
glBegin(GL_POLYGON);
  // Draw verticies in a square
  glVertex2f(0.0, 0.0);
  glVertex2f(0.0, 10.0);
  glVertex2f(10.0, 10.0);
```

```
  glVertex2f(10.0, 0.0);
glEnd();
```

The Java graphics libraries also provide an API for creating graphics[42], this time in an object-oriented form. The programmer has access to a `Graphics` object, which can be passed `Shape`s or coordinates to create polygons, among other features. A simple example where a rectangle is drawn on a "graphics" object could be written as follows:

```
// Create rectangle at point (0,0) of width and height 10
Shape aRectangleShape = new Rectangle(0, 0, 10, 10);

// Tell the graphics object to draw this shape by filling it with color
someGraphicsObject.fill(aRectangleShape);
```

However, doing graphics programming with SVG and ECMAScript is currently quite a different story. In SVG, the programmer expresses the graphics by creating the appropriate elements and attributes in an XML document conforming to the SVG specification. To manipulate the SVG graphics in real-time, code can access and manipulate the SVG document directly via the document object model (DOM). What this means, is that to add a rectangle to the graphic the code needs to create XML elements which describe the shape, and then append the elements as children[1] to appropriate elements in the existing SVG XML document. For example, to dynamically create a square similar to the openGL and Java examples would require writing in a form such as:

```
// Create rect element
var shapeElement = svgDocument.createElementNS(svgns, "rect");

// Set x attribute
shapeElement.setAttributeNS(null, "x", 0);

// Set y attribute
shapeElement.setAttributeNS(null, "y", 0);

// Set width
shapeElement.setAttributeNS(null, "width",  10);

// Set height
shapeElement.setAttributeNS(null, "height", 10);

// Append the rect element to the root element in the SVG XML document
svgDocument.documentElement.appendChild(shapeElement);
```

[1] XML elements inside other XML elements are referred to as the children of the containing element

This illustrates the different approach required in graphics programming with SVG and the DOM versus programming with a typical graphics API. Instead of dealing with vertices and shapes directly, the programmer must deal with the graphics through an extra level of indirection, creating XML elements and attributes, and navigating the XML document tree. Ideally, the programmer should be presented with abstractions of the graphical components themselves. How then can we make accessing SVG from ECMAScript easier?

We can begin by observing that much of the progress in software development is based on building higher-levels of abstraction upon the building blocks that have gone before. We can apply this principle to make the union of SVG and ECMAScript for software visualisation a more attractive proposition. One approach would be to create an ECMAScript graphics API to hide the SVG manipulation, analogous to OpenGL, Java's 2D libraries, or Microsoft's DirectDraw. Alternatively, we could create ECMAScript SVG libraries designed for specific domains. In our case, this would involve creating an ECMAScript library specificly tailored for supporting software visualisation. In section 6.2 I discuss a developing ECMAScript graphics API, and in section 6.3 I detail my development of an ECMAScript SVG library for software visualisation.

6.2 An SVG graphics API

Interestingly, after I started developing an ECMAScript SVG library for software visualisation I discovered some work towards the first approach mentioned, that of building a generic ECMAScript graphics API on-top of SVG[31]. This project includes ECMAScript objects for doing 2D Maths, creating and manipulating SVG Shapes, GUI Widgets such as buttons and sliders, and a number of other utility objects. For example, a button GUI Widget can be created with the following ECMAScript code:

```
button = new Button(
    50, 200,
    my_callback,,button_updy,button_downmb, SVGRoot);
```

`my_callback` is the event handler function that will be called when the button is pushed. `button_updy` and `button_downmb` are XML id values for SVG elements which define the appearance of the button in its two states (up and down). `SVGRoot` is the root XML element in the SVG document. The resulting ECMAScript button object will then abstract away the details of keeping the SVG document up-to-date with the current state of the button, and passing on button events to the appropriate event handlers. Figure 6.1 shows a number of these buttons in use in an SVG document.

Using the functions provided removes the need for navigating the SVG document's DOM and laboriously adding elements and attributes. While this is still happening in the implementation of the ECMAScript objects, the programmer can focus more on their intentions for the

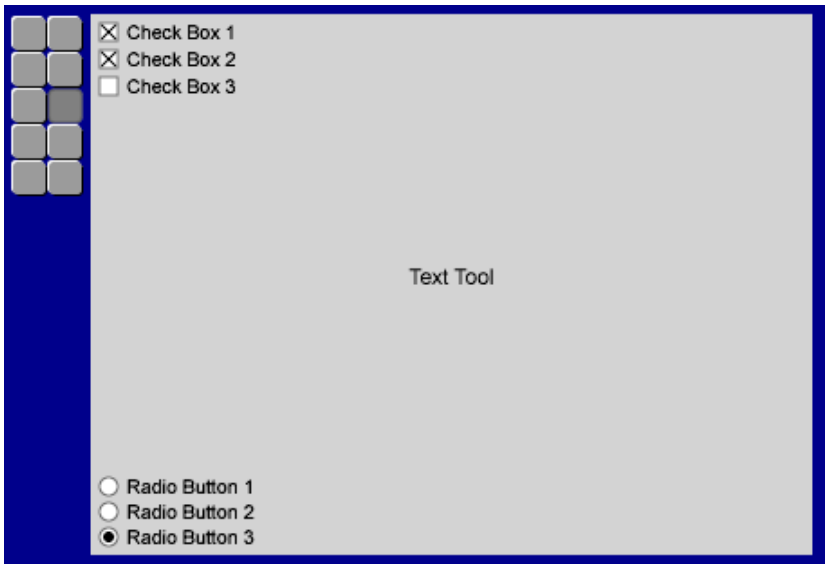


Figure 6.1: A user interface with buttons produced by the KevLinDev[31] SVG ECMAScript library.

graphics themselves, rather than thinking about the structure of the SVG document. However, at the same time, this approach does not try to hide SVG from the programmer entirely. The SVG document is still a very important part of the programming model. For example, the `new Button` function shown above takes references to SVG graphics for the depictions of various button states, rather then passing ECMAScript objects that might encapsulate such graphics. While this is not necessarily a criticism, intuitively it seems less than ideal to force the developer to work with two very different representations of the graphics. It would be interesting to look at this issue further, examining whether there are negative impacts resulting from this added complexity. This is a definite area for future work.

While building a generalised graphics API for SVG has many potential gains, it is also possible to create domain specific ECMAScript libraries.

6.3 An SVG software visualisation library

As described in section 3.1.2, Oudshoorn et al.[47] divide software visualisations into three types: Graph-based (node-link), Statistics-based, and source-code-related displays. If we can augment SVG to better support the development of these sorts of diagrams, SVG could become a much more attractive option for software visualisation deployment. The obvious approach for this would be to create an ECMAScript library that would hide away many of the details of the graphic manipulation of the SVG document, and allow developers to deal directly with objects representing the abstractions they are working with. However, these different types of software visualisation will require very different feature sets from those found in a support library. For example, a node-link type software visualisations would need abstractions for nodes and links, while source-code-related visualisations would require

textual abstractions to deal with large textual structures in intelligent ways.

A practical approach to begin working towards such a library is to develop the foundation of one of the three software visualisation types. Here I describe the development of an ECMAScript library to support the node-link type diagrams. As mentioned in section 5.3.3, SVG is disappointing in its lack of ability to support node-link diagrams. Because SVG has no facility to encode the semantics of a graphic, such as defining a point as being attached to a shape, developers are forced to track events and coordinates and update the SVG accordingly all through ECMAScript. The library aims to alleviate this.

6.4 Library details

The library is designed to be called from a near empty SVG document. An example is shown in figure 6.2. The document only needs to contain two elements, both of which are `script` elements. The first should reference (and load) the node-link ECMAScript library. The second script element contains XML character data, as indicated by the surrounding <![CDATA[and]]> markers. This character data contains the ECMAScript content where the library calls will be made. This ECMAScript contains a function called GenerateStructure() which will be called by the library when the diagram is ready to be built. Finally, the root `svg` element must set the library's Initialiize(evt) function as the event handler for the onload event, which is triggered once the base SVG has been loaded. The Initialize function will set up the required SVG content, create the required resources, and then call the aforementioned GenerateStructure function to generate the diagram.

```
<svg onload="Initialize(evt)" >

  <script type="text/ecmascript" xlink:href="./nodebuilder.js" />

  <script type="text/ecmascript">
    <![CDATA[

      function GenerateStructure(){

        // library calls go here

      }

    ]]>
  </script>

</svg>
```

Figure 6.2: The skeleton SVG file required to use the ECMAScript SVG library.

6.4.1 Creating Nodes and Links

At this point, creating Nodes and links is much simpler with the details of SVG removed. Calls to create these are put in the GenerateStructure function. To build a node, a Node object must be created with the ECMAScript "new" keyword. A number of parameters can be passed to the Node, including its x and y coordinates. The Node object takes care of remembering its location on the SVG canvas, as well as adding itself to the SVG document, and keeping its constituent SVG elements and attributes up-to-date. Creating links between Node objects is equally simple. This involves creating an instance of the Link object with the "new" keyword, passing through the two Node objects to be linked as parameters. Additionally, other parameters, such as the symbols to be drawn at either end of the line can be given. A simple example of creating Nodes and Links can be created by the code in figure 6.3, and the result is shown in figure 6.4.

```
function GenerateStructure(){

  // library calls
  var firstNode = new Node("10","10");
  var secondNode = new Node("80","9");

  new Link(firstNode, secondNode, "none","none");

  var thirdNode = new Node("250","230");
  var fourthNode  = new Node("200","300");

  new Link(thirdNode,fourthNode,"none","arrow");

  var fifthNode = new Node("330","300");

  new Link(thirdNode,fifthNode,"arrow","none");

  var sixthNode = new Node("200","20");

  new Link(sixthNode, fourthNode,"arrow","aggregate");

  var seventhNode = new Node("159", "200");

  new Link(seventhNode, fifthNode, "none","aggregate");

}
```

Figure 6.3: ECMAScript calls creating a dynamic SVG node-link diagram, shown in figure 6.4.

Figures 6.3 and 6.4 show clearly how the library takes care of many details which are awkward to implement in SVG. Firstly, being able to specify links by simply associating them with the nodes which they join, rather than with hard coordinates ensures that all of the nodes

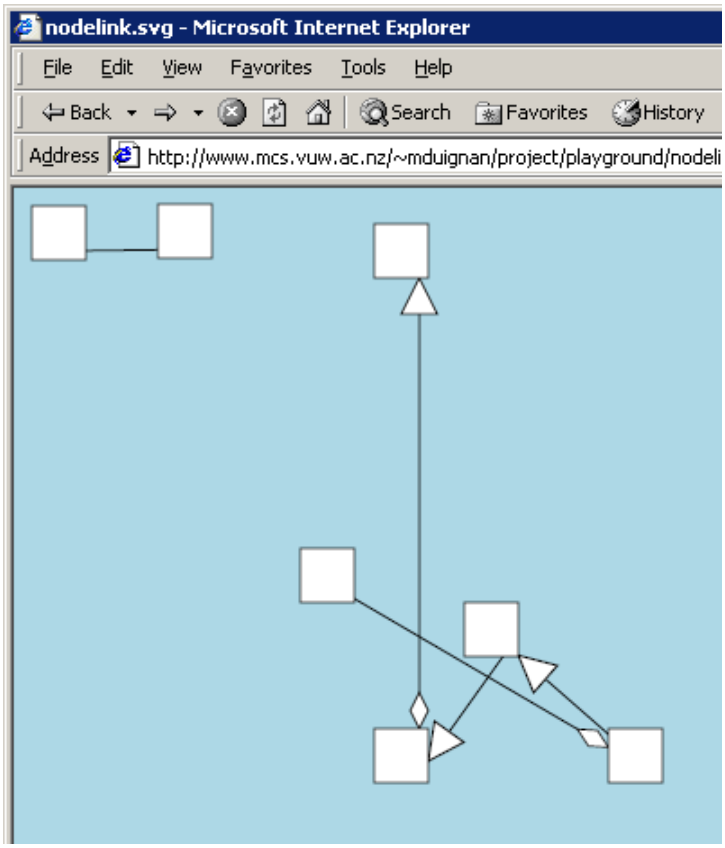


Figure 6.4: A dynamic node-link diagram generated by ECMAScript objects.

and links will be properly connected. The library takes care of ensuring that the symbols at the end of the links are correctly positioned on the edge of the node, and that they have the correct rotation to match the angle of the link.

6.4.2 Dynamic diagrams

While this is clearly helpful, the real benefits can be seen in the dynamic nature of the node-link diagram. What cannot be seen from the simple code example and screen-shot, is that the end-user viewing the node-link diagram can click and drag the nodes to new positions. As a node is being moved, the attached links move with it, and the symbols on the links also move and rotate as appropriate. This can be seen in action in figure 6.5.

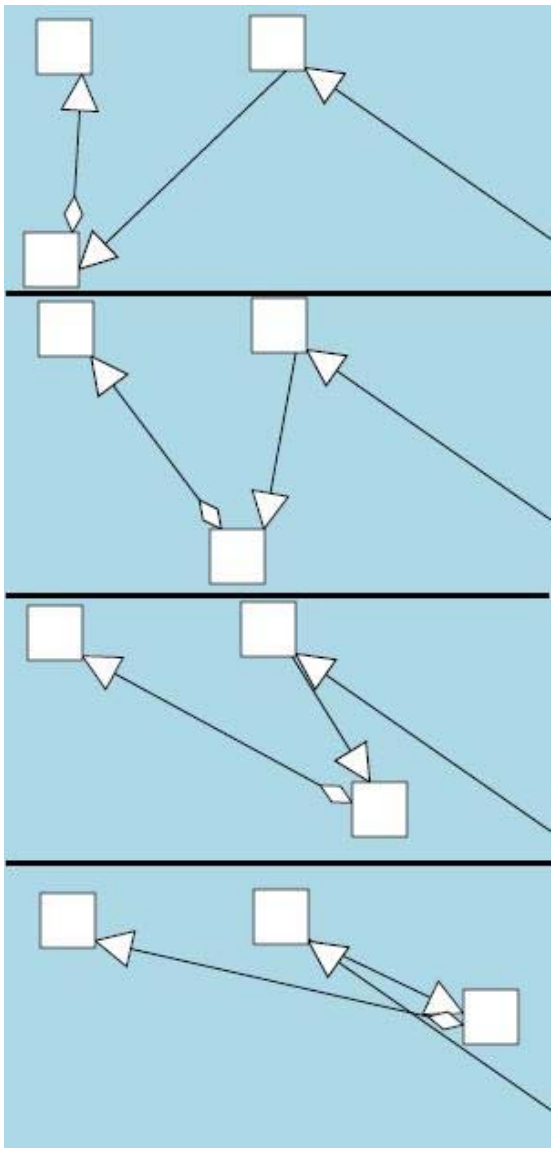


Figure 6.5: Four shots of a Node in a node-link diagram being dragged from left to right by the mouse.

This is made possible as the Node object has registered itself in the SVG document as the

recipient of mouse events occurring over the square box depicting the Node. When there is a mouse event (such as a move or click) over the Node's depiction in the SVG graphic, the Node object is alerted, and it can then update the appropriate elements in the SVG document as necessary. As the Node object moves, it also advises the Link object (via LinkEnd objects) of its changing position, so the links follow as well, updating their own collection of elements in the SVG document.

Each object that has a depiction in the SVG document has a refresh() method, which contains all of the code that is responsible for updating the SVG elements. This ensures that all of the SVG specific code is encapsulated in the single method, and changing the appearance of an object only requires modification of this method.

As is probably evident, the node-link functionality is essentially useless for software visualisation unless the appearance and behaviour of the nodes and links can be customised. The ECMAScript GUI library mentioned in section 6.2 allowed the specification of new graphical content for widgets by passing a reference to new SVG content contained in the SVG document. However, this approach is not taken here for two reasons. Firstly, this only allows the specification of new graphical appearance for the objects, not behaviour. New non-trivial behaviours cannot be added without working with ECMAScript. Secondly, using SVG at this level forces the developer to deal with SVG directly every time they wish to instantiate an object.

Another approach which avoids both of these problems is to encapsulate SVG content in the object-oriented nature of ECMAScript.

6.4.3 Object-oriented ECMAScript

While ECMAScript does not contain all of the features of a full object-oriented programming language, it does have a number of the important object-oriented features. ECMAScript provides the capability to instance objects with the "new" operator, in much the same way as with Java. These objects contain data, and have attributes and methods which allow the user of the object to access and manipulate this data. However, unlike Java, ECMAScript is a form of "Prototype-based language"[17]. The "new" operator simply creates an empty object — calling a constructor function to build the object by adding attributes and methods to the object as required. Additionally, any object can be used as a prototype for a new object. Any attributes or methods belonging to the prototypical object will also be available in the new object. Additional attributes and methods can be added to either object at anytime during the program's execution. Additionally, an object that has has a prototypical parent can override any methods with its own.

6.4.4 A UML class node

This overriding functionality can be used to create new versions of objects to use in the library, with the rest of the library remaining unaware of the changes. In our case, this

means that new appearances, behaviour and functionality can be added to Node objects and the library will continue to manipulate them as normal.

For an example, I developed a basic interactive SVG UML Class diagram based on the library. This simply involved creating a new type of Node, or more correctly, creating a UML Class object that is prototyped from an existing Node object. The code to accomplish this can be contained in the script element inside the SVG document, or could be put in an external ECMAScript file and stored on the Web for reuse. The actual implementation requires creating two new functions. The first is the Objects constructing function, and the second is a "refresh" function which will override the "refresh" method of the Node object from which it is prototyped. The refresh method contains all of the logic required to update the SVG document with the objects current state. These functions are then bound to the UML Class object as methods, and the Node object is specified as the prototype.

Of course, a UML Class node is different from a plain Node in that it has a name, attributes and methods. To implement a UML Class node, extra methods can be added to the ECMAScript UML Class node, such as addMethod(*methodName*) and addAttribute(*attributeName*) methods. Once the refresh() method has also been enhanced to add these methods and attributes to the SVG document, a new UMLClass can be created from within an SVG document's script content. Figures 6.6 and 6.7 show a UMLClass object created by the ECMAScript library.

```
function GenerateStructure(){

  // library calls
  var classNode = new UMLClass("10","10", "SomeClass");
  classNode.addMethod("Method1()");
  classNode.addMethod("Method2(anArgument)");
  classNode.addMethod("Method3(anotherArg)");
}
```

Figure 6.6: ECMAScript calls creating a UMLClass node, shown in figure 6.7.

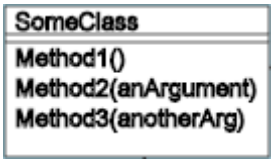


Figure 6.7: A UML Class node created by the SVG ECMAScript library.

Because the UMLClass objects are created from Node objects, they can be used in library calls as Nodes could before. This means that links can be created between the UMLClasses by passing the library's Link objects the appropriate UMLClasses to link. Any sort of Nodes can be linked together, so new Nodes, such as documentation notes, can be created and linked with other Nodes in the diagram.

6.5 SVG node-link library in action

With these objects available from within the SVG script content, it is trivial to create UML Class diagrams. Figure 6.8 shows the ECMAScript code required to generate the SVG class diagram shown in figure 6.9. Note that the user can click and drag the classes around the diagram to new positions as desired. This functionality comes from the fact that the UMLClass objects are prototyped from the Node object.

```
function GenerateStructure(){

  // library calls
  var car = new UMLClass("100","10", "Car");
  car.addMethod("startEngine()");
  car.addMethod("drive(Road)");

  var bmw = new UMLClass("20","130","BMW");
  bmw.addMethod("activateCarAlarm()");

  var road = new UMLClass("250","10", "Road");
  road.addMethod("getCondition()");

  var electricWindows = new UMLClass("200","230", "ElectricWindows");
  electricWindows.addMethod("open()");
  electricWindows.addMethod("close()");


  new Link(car, bmw, "arrow", "none");

  new Link(car, road, "none", "none");

  new Link(bmw, electricWindows, "aggregate", "none");

}
```

Figure 6.8: ECMAScript calls creating a UMLClass diagram, shown in figure 6.9.

With this ECMAScript library available, it greatly simplifies the process of generating these diagrams programmatically on request. To experiment with this, I augmented Blur to create SVG interactive class diagrams from PAL files. From a combination of the static type information, and dynamic runtime events contained in PAL, it is possible to create the appropriate SVG ECMAScript library calls and embed them in an SVG document. When this SVG document arrives at a client, the ECMAScript library will create the diagram, as specified by the embedded library calls. The user can then manipulate the diagram as they require. After integrating this into the web interface discussed in section 4.3.2, it is now possible for users to request interactive UML class diagrams to be built from any PAL output, and have it deployed in their browser as SVG.

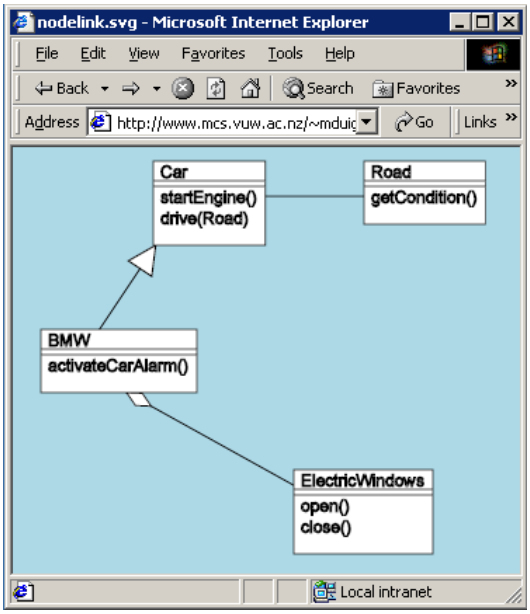


Figure 6.9: A UML Class diagram created by the SVG ECMAScript library.

Instead of building another layout algorithm implementation from scratch, I incorporated the graph layout system from the Graph Visualization Framework (GVF)[37]. The GVF provides graph and layout functionality that allows new layout algorithms to be plugged in. Blur uses the Fruchterman Reingold layout algorithm[23] as implemented by GVF. The initial layout is calculated on the server, and the calculated coordinates are used in the ECMAScript library calls embedded in the SVG document. Due to random elements in this layout algorithm, the user can press the web browsers refresh button, and cause a new layout to be generated on the server and displayed in the browser. Figure 6.10 shows four layouts of a class diagram created from the same PAL file. Once the user finds a layout they are reasonably happy with, they can fine tune the layout by clicking and dragging the classes with their mouse.

Figure 6.11 shows another UML class diagram created via the web interface from a PAL output file. The PAL file used was created by a prototype Java engine for the VARE framework. This engine is similar to the engine "AT" that was discussed in section 4.3.1, but rather than generating PAL from the execution of C++ programs, it builds PAL from the execution of Java programs, by using the Java Debugger Interface (JDI)[40]. Interestingly, in this case the Java program used as the source for the PAL output was a previous execution of Blur itself. When this PAL file describing an execution of Blur, was fed back into Blur, it was able to draw software visualisations of itself. While the information in this diagram is incomplete, it shows how the ECMAScript SVG software visualisation library can already be used to create reasonably detailed diagrams.

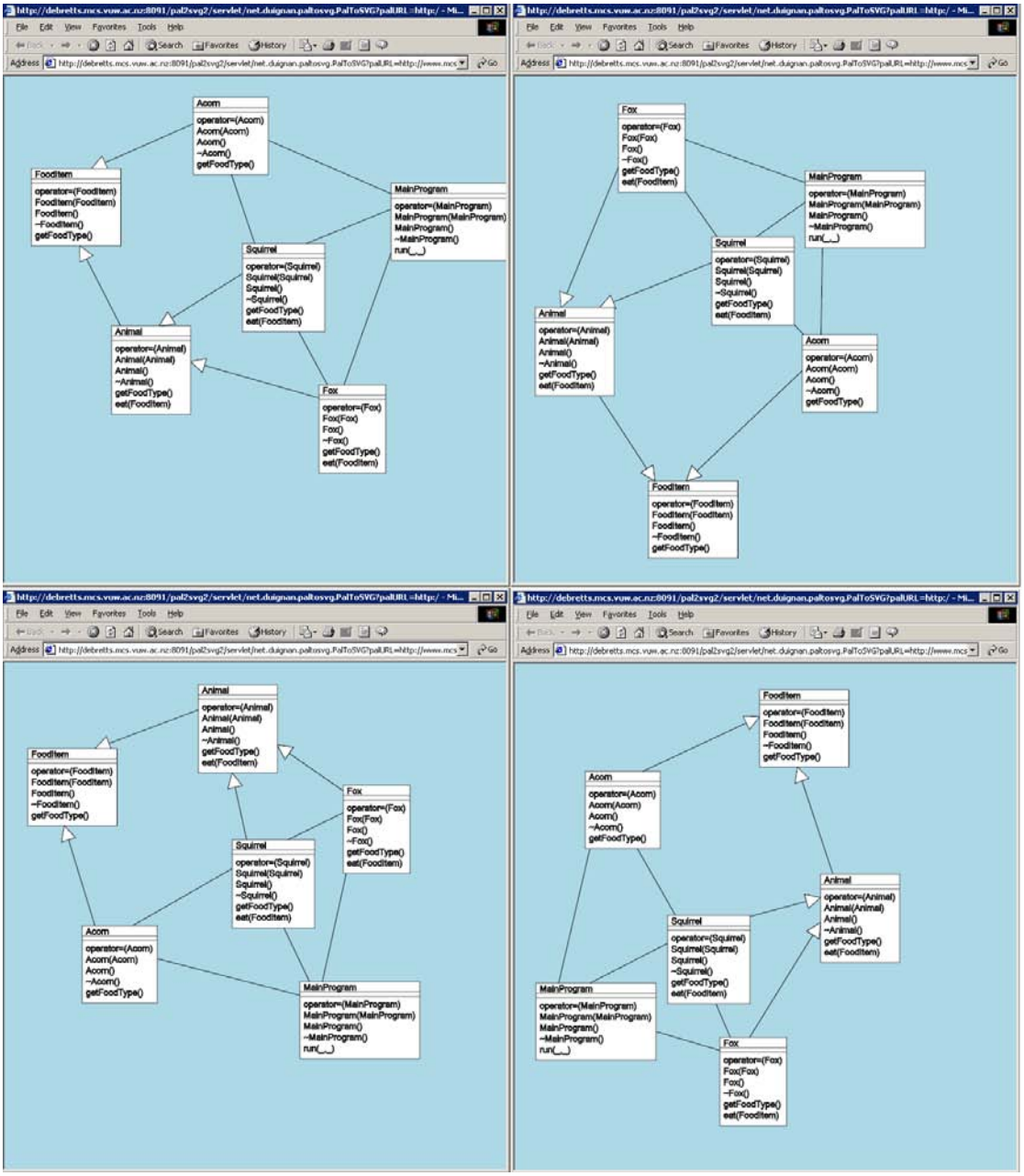


Figure 6.10: Four different layouts of a SVG UML class diagram.

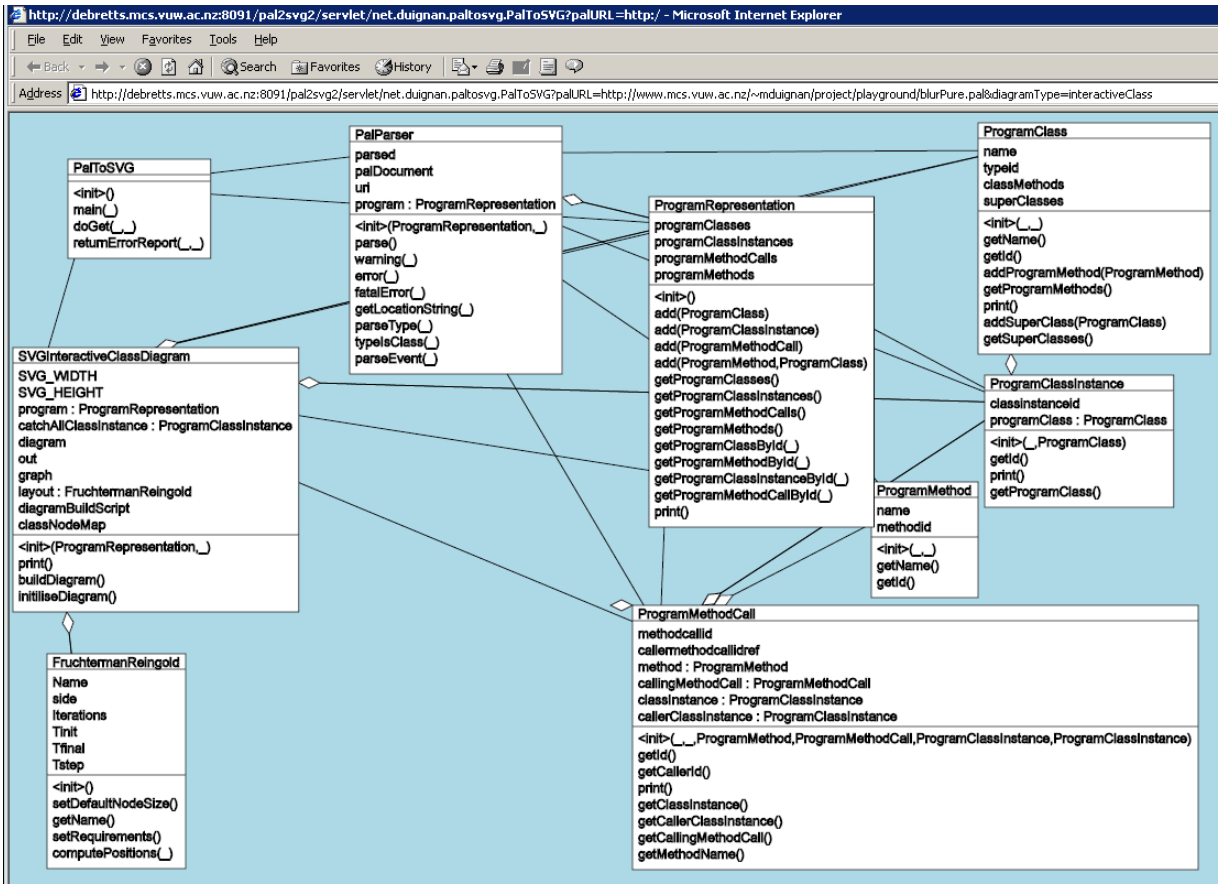


Figure 6.11: A UML class diagram of Blur, created by Blur from a previously created PAL file.

6.6 Discussion

While this is only the beginning of what is required for a full SVG software visualisation library, there are a number of interesting issues that its development has highlighted.

By using a clear ECMAScript interface to create the SVG document, it is possible to ensure that all SVG specific code is kept in the same environment. In contrast, the SVG content of the collapsible sequence diagram shown in figure 4.21 was created on a server using Java. However, the code which modifies the diagram to enable the interaction was implemented in ECMAScript and runs on the client system in the web browser. This means that the functionality which is dependent on the structural details of the SVG diagram is spread over the client and server, in two different development environments. This makes any maintenance of the diagram difficult, as both the Java code on the server, and the ECMAScript on the client will need to be carefully modified in tandem. This situation can now avoided by using the SVG software visualisation library. Using only a clean ECMAScript library as I have for the UML class diagram generator, means that all the SVG generation and modification for interaction is centralised. This eases the maintenance and development of the SVG diagrams. However, as was discussed in section 5.3.3 this comes at a price. Having so much depending on client-side scripting can complicate matters. Scripting across multiple web-browsers on multiple operating systems has traditionally been a point of difficulty due to the variations in script engine implementations. This difficulty will only be exacerbated by the fact that SVG demands no one standard scripting environment.

Another issue that arose in implementing this node-link library is that while it attempts to isolate the developer from the details of SVG while building node-link diagrams, it is still necessary to use SVG to generate new object types. For example, implementing the UML Class object required writing a refresh() method which manipulated SVG content. If the idea of hiding SVG's implementation was taken to the extreme, it would be possible to create a library of ECMAScript drawing primitives to manipulate the SVG. This could provide the developer with a more coherent development environment. More investigation is needed to determine what the real benefits of this would be. Additionally, the added complexity introduced in the ECMAScript could cause additional performance problems.

6.6.1 Possible improvements

While this ECMAScript SVG node-link diagram library is only the beginning of a full library, there are a number of small additions which would add greatly to the current implementation's utility.

One big improvement to the Class diagram generator would be to use a specialised class diagram layout algorithm. A Java framework is available[18] designed specificly for this task providing an implementation of one such algorithm. Such an algorithm would give better end results, as more general algorithms have no knowledge of the semantics of UML, and thus cannot make intelligent layout decisions. Many of these more sophisticated algorithms

require the ability to specify links made up of multiple points. Again, this would simply require creating a new object for the library which is prototyped from the existing Link object. This would take a series of points as an argument, and draw the appropriate SVG on a refresh() call. Users would expect to be able to add, delete, or move the points in a link.

Another important addition would be to include the functionality for the user to save the diagram after they have made modifications to the layout. This could be done by saving a static snapshot of the diagram as non-dynamic SVG, or by recording the new layout coordinates so new library calls could be generated to create the diagram from scratch. Ideally, these coordinates should be able to be saved back to the web-server which provided the diagram, for future viewings. However, the SVG standard does not specify any form of network communication that SVG viewers must support. This means that such functionality would need to utilise individual SVG viewers' proprietary communication extensions. The Adobe SVG Viewer provides a getURL method that can be accessed from the script content which could be used for such a purpose.

6.6.2 Contributions

This chapter has illustrated the weaknesses of SVG when viewed as a development environment for interactive graphical content, and how these can be addressed through scripting. In particular, when viewed in this light, it is interesting how it compares to a programming language such as Java which provides an integrated mechanism for both displaying graphical objects, and supporting the computation necessary for interaction with these objects. In Java, developers create graphical content in exactly the same way that they then modify it for interaction — by writing *Java* source-code.

This chapter showed how it is possible with some effort, to use SVG's scriptability in building a resource to bring SVG up to some more acceptable level. By building the foundation of a ECMAScript SVG software visualisation library which can be linked to dynamically by SVG documents, it is now clear that to a large extent, developers *do not* need to deal with the file format details of SVG when creating standard types of interactive software visualisation. By hiding most of the details of the SVG implementation under the ECMAScript library, developers can work with just ECMAScript, and use its object-oriented features to work with graphical "objects" in a more high-level manner.

Chapter 7

Conclusions

The goal of this thesis was to evaluate SVG for use in software visualisation. In particular, a technology like SVG is required for our research group's developing Visualisation Architecture for Reuse (VARE). To conduct a thorough evaluation of SVG required developing an evaluation model for software visualisation media.

In this thesis I have followed a clear methodology. In chapter 3 I developed a principled model of the required capabilities for software visualisation media. In chapter 4 I showed the basis for learning SVG's basic capabilities, and described the SVG software visualisations created by me, for the purpose of software visualisation specific evaluation. In chapter 5, the developed model, coupled with the knowledged gained from exploring SVG, was put to use in evaluating SVG's capabilities for software visualisation. In chapter 6 I examined the possibilities for extending SVG by supporting it with ECMAScript libraries, and described my development of the beginnings of an ECMAScript library for SVG software visualisation. This thesis also outlines the contribution of a transformer for the VARE architecture, as well as assessing the suitability of SVG for inclusion in its implementation.

From these results, SVG is clearly capable of displaying at least a set of useful software visualisations. To conclude, I will comment on the consequences of this work for the VARE architecture, the success of the evaluation model, and look at potential work for the future.

7.1 Conclusions for VARE

The primary motivator of this investigation was to evaluate SVG for use in the VARE architecture. Therefore it is vital to discuss the impact the results of our evaluation have on VARE.

7.1.1 Contribution to VARE

A fully functional PAL to SVG software visualisation transformer called Blur was implemented for UML collaboration diagrams, collapsible sequence diagrams, and interactive class

diagrams. With the implementation of a SOAP control layer, these could be integrated into VARE as soon as is necessary. The development of Blur proves that PAL is a usable format for encoding at least the basic set of process abstractions for visualisation. The construction of Blur allowed the complete PMV process to be carried out in VARE from beginning to end. A real, albeit simple program, was written in C++. It was run under AT. PAL output was taken and fed into Blur, and software visualisations were created that were deployed over the web. To illustrate the platform neutral capabilities of the architecture, a Java program (which was a previous execution of the transformer Blur itself) was also integrated, this time by a Java specific engine. The resulting PAL was fed into Blur and software visualisations were created. Due to the current lack of coordinated control in VARE, some hand-holding was necessary, but this clearly provides an important proof of concept for the basis of the VARE architecture.

7.1.2 Implications for VARE

While this looks generally positive for the current direction of VARE, the results of this evaluation point to a number of conclusions that should influence VARE's architecture.

Implications for PAL

While PAL is functional as an encoding for process execution information, it is not particularly friendly to work with. This, as well as the tree-traversal distraction of the DOM, combine to make PAL a non-ideal representation for use when creating visualisations. This means that creating an in-memory representation of the process execution described in a PAL document will be an important aspect of almost all transformers. The implication of this is that VARE needs a standard and convenient way to represent process executions in memory. If this was developed, moving from one transformer to another for maintenance or creation would be eased by a shared design. What's more, a convenience library could be written for popular transformer languages (perhaps Java and Python) which would take care of this common functionality.

This is not an indication of a weakness in PAL. Much of the awkwardness in navigating PAL via the DOM is due to the side effects of good data design rather than a usability oversight. PAL's data format is modular and it makes heavy use of XML's IDREF attribute type to maintain this. A PAL element representing a method-call event will only have information specific to that method-call. All shared information such as the information on the method being called is accessible through the reference provided by an IDREF. The IDREF refers to a unique element with a matching ID attribute. To gain some information, such as the object type that a particular method-call is called on, requires following several IDREF to ID jumps. This helps ensure that a PAL document will be internally consistent. This is because it is more difficult to make mistakes in PAL generation as each piece of information

is only located in one place. This also reduces the size of PAL output because of the lack of duplicated information.

However, in PAL's current, still evolving form, I found a break in its adherence to this pattern. PAL's specification states that a method-call event element can have a callerclassinstanceidref. This reference to the object which is calling the method on another object, is also available by following the callermethodcallidref, and finding the object from that element instead. While this is a minor incongruity, it was already causing confusion with AT's output. All that this indicates is that PAL's specification needs to be carefully reexamined and (to borrow a term from databases) be "normalised".

There is also an interesting issue due to PAL's nature as a pure description of type and event data. There are characteristics of program design, such as aggregation and structures like loops that PAL cannot clearly capture. While this is often information that it may be difficult or impossible to simply deduce from interrogating a programs execution, it is important that these factors are looked at for future development of PAL. For example, a loop in a program which called a method several hundred times would be represented in PAL as several hundred method-calls. A visualisation of this would be much more intelligible to a human if it somehow represents the loop itself, rather than just a huge number of method-calls.

Using more than one visualisation medium

The problem with SVG seems to be that it it is a little too light-weight. Java on the other hand, is too heavy weight. This tends to suggest that some software visualisations might best be served by SVG, while more complex visualisations might have to be implemented in Java. This would provide the best of both worlds, but would have a number of consequences.

- The client-side parts of the architecture would need to be designed to allow both SVG and Java to be used.
- VARE's "Visualisation Repository" would need to be able to store software visualisations implemented in both Java and SVG, or Java applications would need to be able to display the smaller SVG visualisations.
- A protocol for multiple view coordination would need to be established that was interoperable between Java and SVG.

7.2 Meta analysis — evaluating the model

Because I developed a new model for the evaluation, it is important to reflect on it briefly here. Having a model for evaluation gave a clear guide to what was important in visualisation. Having such a guide was important to ensure that no necessary capabilities were ignored in the evaluation. The model's list-like form seemed to be best utilised by commenting briefly

on how a medium matched up to the identified capabilities, and then pulling out the more interesting observations for further discussion. While some of the model was conceptually new work in bringing together these important capabilities, it seemed to be successful on its first try. No doubt, more capabilities could be added, and details need to be refined, but the basic form is already highly usable.

7.3 Summary of evaluation

In summary, SVG definitely has potential for use in the VARE architecture. As shown by figure 4.5, SVG is already being used for software visualisation, and it is clearly capable of simple to medium sized visualisation demands. However, SVG is disappointingly low level in its core functionality. The integration of scripting addresses this to an extent by giving it the capability for more complex visualisations. On the other hand, the reliance on scripting for all of this more complex work makes SVG more awkward and undesirable than it needs to be. Inclusion of layout constraints would instantly give SVG more credibility for serious use in software visualisation. As this is not likely to happen in the near future, the next best bet is to carefully develop an SVG software visualisation scripting library. Our explorations of this solution proved that this would be a useful endeavour.

SVG is a good choice for light-weight visualisations. It is quick and easy to deploy over the web, and it provides high quality graphical capability. However, more complex and interactive visualisations would be better served by some other medium — such as Java Applets.

7.4 Future work

This thesis has illuminated a number of possibilities for future research. The software visualisation media evaluation model could be further developed by looking at non-visual capabilities such as sound and touch, or by expanding and refining the high-level capabilities. It would also be useful to look at conducting evaluations with the model on other potential software visualisation media, such as Flash, X3D, VRML and Java. This would increase the utility of the SVG evaluation presented in this thesis, as well as testing the applicability of the model to visualisation media other than SVG.

In terms of further research into SVG, it would be useful to study the effects of the dichotomy between SVG development, and ECMAScript development. Because developers are expected to create some SVG content directly in XML, and then manipulate and create other SVG content from an imperative language such as ECMAScript, this results in an inconsistent and perhaps overly complex development process. It would be valuable to study the effects of this on program understanding, and developer productivity.

Finally, this thesis raised some issues for the VARE architecture which invite further research. Firstly, VARE would gain from the development of a useful standard for in-memory program

representation. This would provide a useful and standard abstraction for programmers building transformers under VARE. It would be interesting to look at ways to embed this into the VARE architecture, perhaps by making such functionality into a library, or by making it part of the API for an XML PAL database. Also, if SVG is to be used in VARE alongside other visualisation media, it may be important to look into issues of view-coordination inside VARE visualisations.

7.5 Contributions

This project achieved the following:

- A principled model for evaluating software visualisation media was developed and tested. This model allows software visualisation developers to assess the strengths and weaknesses of a medium in a methodological and thorough manner. The model also provides the ideal foundation for the comparison of potential software visualisation media.
- SVG was carefully evaluated as a medium for software visualisation and its core strengths and weaknesses presented. This evaluation can be used by software visualisation developers who are considering using SVG as a display technology. The evaluation makes clear how SVG performs in various software visualisation capabilities which may be required by developers for specific software visualisation tasks.
- A method for proceeding with using SVG as a medium for software visualisation was presented, namely through the creation of a specialised scripting library. It was shown how such a method would provide commonly required functionality while hiding the low-level nature of SVG.
- This solution was tested by implementing the beginnings of such a library. This library can now be used for the creation of node-link diagrams — and vastly simplifies their development. It was shown by example how this library can be easily extended to support new node-link software visualisations. This library not only simplifies the development of these visualisations, but also provides additional built-in functionality such as allowing users to interactively manipulate the visualisation once it has been rendered.
- The implications of the evaluation on VARE were discussed. It was shown how the Process Abstraction Language (PAL) is not the best format for visualisation developers to deal with. Instead, PAL should be used as the underlying file format, and automatically turned into a more user-friendly object model via a standard API. It was also shown that if the VARE architecture settles on SVG for its primary visualisation medium, it should ensure that it also has the ability to support other media (such as Java) for more complex and demanding visualisations.

- A live demonstration of VARE in action was conducted, showing an important proof of concept for the VARE architecture. It is now clear that the underlying principles of the VARE architecture are sound. The completion of the SVG visualisation transformer for VARE was the final element required to show the process of a code component being executed, interrogated, and visualised — all conducted remotely over the Web. This transformer can now be easily integrated into the final VARE architecture to provide a number of SVG software visualisations.

Bibliography

[1] R. M. Baecker, B. A. Price, and I. S. Small. A principled taxonomy of software visualisation. *Journal of Visual Languages and Computing*, 4(2):211–266, September 1993.

[2] B. B. Bederson and J. D. Hollan. ad++: A zooming graphical interface for exploring alternate interface physics. In Marina del Rey, editor, *Proceedings of UIST'94, ACM Symposium on User Interface Software and Technology*, pages 17–26, California, USA, November 1994.

[3] J. Bertin. *Semiology of Graphics: Diagrams, Networks, Maps*. University of Wisconsin Press, Madison, WI, 1967/83. W. J. Berg, Trans.

[4] R. Biddle, M. Duignan, K. Jackson, S. Marshall, M. McGavin, and E. Tempero. Visualising reusable software over the web. In *Information Visualisation 2001. Australian Symposium on Information Visualisation*, volume 9, pages 103–111, 2001.

[5] D. Brownell. Sax. `http://sax.sourceforge.net/`.

[6] S. Burbeck. Applications programming in smalltalk-80: How to use model-view-controller (MVC). `http://st-www.cs.uiuc.edu/users/smarch/st-docs/mvc.html`.

[7] S. K. Card, J. D. Mackinlay, and B. Shneiderman. *Readings in Information Visualisation: Using Vision to Think*. Morgan Kaufmann, 1999.

[8] International Color Consortium. Document icc.1a:1999-04 addendum 2 to spec. icc.1:1998-09. `http://www.color.org/ICC-1A\_1999-04.PDF`.

[9] The Web3D Consortium. Extensible 3D (X3D) graphics. `http://www.web3d.org/x3d.html`.

[10] The Web3D Consortium. The virtual reality modeling language. `http://www.web3d.org/Specifications/VRML97/`.

[11] Rational Software Coperation. UML quick reference for rational rose. `http://www.rational.com/uml/resources/quick/index.jsp`.

[12] Corel. Corel smart graphics. `http://www.corel.com/smartgraphics`.

[13] K. C. Cox and G. Roman. A taxonomy of program visualization systems. *IEEE Computer*, 26(12), December 1993.

[14] Objects By Design. Open source. `http://opensource.objectsbydesign.com`.

[15] H. A. D. do Nascimento. A framework for human-computer interaction in directed graph drawing. In *Information Visualisation 2001. Australian Symposium on Information Visualisation*, volume 9, pages 63–69, 2001.

[16] T. Dwyer. Three dimensional UML using force directed layout. In *Information Visualisation 2001. Australian Symposium on Information Visualisation*, volume 9, pages 103–111, 2001.

[17] ECMA. Standard ecma-262. ecmascript language specification. 3rd edition (december 1999). `http://www.ecma.ch/ecma1/STAND/ECMA-262.HTM`.

[18] Holger Eichelberger. Sugibib. automatic layout in UML and case. `http://www2.informatik.uni-wuerzburg.de/staff/eichelberger/SugiBib/engl%ish.html`.

[19] S. G. Eick, J. L. Steffen, and E. E. Summer. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[20] The Apache Software Foundation. Batik svg toolkit. `http://xml.apache.org/batik/`.

[21] The Apache Software Foundation. Xerces java parser. `http://xml.apache.org/xerces-j/index.html`.

[22] M. Fowler. *UML Distilled Second Edition: A Brief Guide to the Standard Object Modeling Language*. Addison Wesley Longman, Inc., USA, 1999.

[23] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software Practice and Experience*, 21(11):1129–1164, 1991.

[24] The Object Management Group. Unified modeling language (UML), version 1.4. `http://www.omg.org/technology/documents/formal/uml.htm`.

[25] XML Hack. Microsoft office embraces xml. `http://www.xmlhack.com/read.php?item=1839`.

[26] D. F. Jerding and J. T. Stasko. The information mural: A technique for displaying and navigating large information spaces. In *Proceedings of the IEEE Visualization '95 Symposium on Information Visualization*, pages 43–50, Atlanta, GA, October 1995.

[27] KDE. KSVG. `http://svg.kde.org/`, 2002.

[28] E. Kraemer and J. T. Stasko. The visualisation of parallel systems: An overview. *Journal of Parallel and Distributed Computing*, 18:105–117, 1993.

[29] Rob Lanphier. svg2swf. http://freshmeat.net/projects/svg2swf/.

[30] J. Larking and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11(1):65–99, 1987.

[31] Kevin Lindsey. Kevlindev. http://www.kevlindev.com.

[32] Kevin Lindsey. pSVG. http://www.kevlindev.com/dom/pSVG/index.htm.

[33] J. D. Mackinlay. *Automatic Design of Graphical Presentations*. PhD thesis, Stanford University, 1986.

[34] Macromedia. Macromedia flash 5. http://www.macromedia.com/software/flash/.

[35] Macromedia. Macromedia flash file format (swf) sdk. http://www.macromedia.com/software/flash/open/licensing/fileformat/.

[36] Macromedia. Xml transfer and html text support. http://www.macromedia.com/software/flash/productinfo/features/new_featu%res_tour/14xml_html.htm.

[37] M. S. Marshall, I. Herman, and G. Melançon. An object-oriented design for graph visualization. *Software Practice and Experience*, 31(8):739–756, 2001.

[38] Mike McGavin. Extracting software re-use information for visualisation tools. Honours report, 2001. Victoria University of Wellington.

[39] Sun Microsystems. Java applets. http://java.sun.com/applets/index.html.

[40] Sun Microsystems. Java debug interface. http://java.sun.com/products/jpda/doc/jdi/.

[41] Sun Microsystems. Java servlet technology: The power behind the server. http://java.sun.com/products/servlet/index.html.

[42] Sun Microsystems. Package java.awt. http://java.sun.com/j2se/1.4.1/docs/api/java/awt/package-summary.html.

[43] Ming. Ming - an swf output library and PHP module. http://www.opaque.net/ming/.

[44] Mozilla.org. Mozilla svg project. http://www.mozilla.org/projects/svg.

[45] J. Noble. *Abstract Program Visualisation: Object Orientation in the Tarraingim Program Exploratorium*. PhD thesis, Victoria University of Wellington, 1995.

[46] OpenGL.org. OpenGL - high performance 2d/3d graphics. http://www.opengl.org/.

[47] M. J. Oudshoorn, H. Widjaja, and S. K. Ellershaw. Aspects and taxonomy of program visualisation. In Peter D. Eades and Kang Zhang, editors, *Software Visualisation*, volume 7, pages 3–26. World Scientific, Singapore, 1996.

[48] T. Pattison and M. Phillips. View coordination architecture for information visualisation. In *Information Visualisation 2001. Australian Symposium on Information Visualisation*, volume 9, pages 165–171, 2001.

[49] W. De Pauw, R. Helm, D. Kimelman, and J. Vlissides. An architecture for visualising the behavior of object-oriented systems. In *OOPSLA 1993*, pages 326–337, 1993.

[50] RFC. Multipurpose internet mail extensions: Mime, rfcs 2045 - 2049. `http://www.rfc-editor.org`.

[51] Gruia-Catalin Roman and Kenneth C. Cox. Program visualization: The art of mapping programs to pictures. In *International Conference on Software Engineering*, pages 412–420, May 1992.

[52] J. T. Stasko. The path-transition paradigm: a practical methodology for adding animation to program interfaces. *Journal of Visual Languages and Computing*, 1(3), September 1990.

[53] Adobe Systems. Adobe illustrator 10. `http://www.adobe.com/products/illustrator/main.html`.

[54] Adobe Systems. Inspiration: Demos. `http://www.adobe.com/svg/demos/main.html`.

[55] Adobe Systems. Svg zone. `http://www.adobe.com/svg/basics/intro.htm`.

[56] J. J. Tirtowidjojo, K. Marriott, and B. Meyer. Extending SVG with constraints. In *Sixth Australian World Wide Web Conference*, 2000.

[57] World Wide Web Consortium (W3C). Cascading style sheets, level 1. W3C recommendation 17 december 1996, revised 11 january 1999. `http://www.w3.org/TR/REC-CSS1`.

[58] World Wide Web Consortium (W3C). Document object model (dom) level 2 core specification. W3C recommendation 13 november, 2000. `http://www.w3.org/TR/2000/REC-DOM-Level-2-Core-20001113`.

[59] World Wide Web Consortium (W3C). Extensible markup language (xml). `http://www.w3.org/XML`.

[60] World Wide Web Consortium (W3C). Extensible markup language (xml) 1.0 (second edition). W3C recommendation 06 october 2001. `http://www.w3.org/TR/2000/REC-xml-20001006`.

[61] World Wide Web Consortium (W3C). Scalable vector graphics (SVG) 1.0 specification . W3C recommendation 04 september 2001. `http://www.w3.org/TR/2001/REC-SVG-20010904/`.

[62] World Wide Web Consortium (W3C). Scalable vector graphics (SVG) 1.1 specification . W3C proposed recommendation 15 november 2002. `http://www.w3.org/TR/SVG11/`.

[63] World Wide Web Consortium (W3C). Scalable vector graphics (SVG) 1.2 . W3C working draft 15 november 2002. http://www.w3.org/TR/SVG12/.

[64] World Wide Web Consortium (W3C). Soap version 1.2 part 0: Primer W3C working draft 17 december 2001. http://www.w3.org/TR/soap12-part0/.

[65] World Wide Web Consortium (W3C). Synchronized multimedia integration language (smil 2.0) . W3C recommendation 07 august 2001. http://www.w3.org/TR/smil20/.

[66] World Wide Web Consortium (W3C). Xforms 1.0 . W3C candidate recommendation 12 november 2002. http://www.w3.org/TR/2002/CR-xforms-20021112/.

[67] World Wide Web Consortium (W3C). Xhtml 1.1 - module-based xhtml . W3C recommendation 31 may 2001. http://www.w3.org/TR/xhtml11/.

[68] World Wide Web Consortium (W3C). Xml events. an events syntax for xml . W3C working draft 12 august 2002. http://www.w3.org/TR/xml-events/.

[69] C. Ware. *Information Visualization: Perception for Design.* Morgan Kaufmann, San Fancisco, 2000.