UNIVERSITY OF SOUTHAMPTON

TREE-SHEETS AND STRUCTURED DOCUMENTS

By
Thomas Leonard
B.Sc.(Hons)

A thesis submitted for the degree of
Doctor of Philosophy

Department of Electronics and Computer Science,
University of Southampton,
United Kingdom.

July 6, 2004

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING

ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

TREE-SHEETS AND STRUCTURED DOCUMENTS

by Thomas Leonard

For data to be used effectively, it needs to be structured and annotated to make it amenable to machine processing. The Web Consortium's XML markup language promises to allow this, and increasing quantities of data are now available in the format. XML's support for namespacing allows information from diverse data sources to be combined and communicated reliably, making XML the keystone of the emerging Semantic Web.

Rather than searching the web by giving a number of text strings which pages must contain, as is common now, this semantically enriched web will allow searches on precisely specified concepts, identified by the page's meta-data. Inferencing engines can be applied to this web of formally expressed facts, so that searches return information which has been deduced from the knowledge on the web, without it having been stated explicitly.

There is a strong need for tools that provide flexible ways to create and manipulate XML documents. Just as the usefulness of having scientific data available in computer-readable CSV (*comma separated values*) format is greatly enhanced by the existence of the spreadsheet, so this new XML-based data requires a new general purpose end-user processing tool. XML is a tree-structured format, and we propose in this thesis the *tree-sheet*: a general purpose end-user tool for the manipulation of tree structured data.

In this thesis we present Dome, our implementation of a tree-sheet. Dome is operated through a familiar direct manipulation style interface, with a 'record' mode being used to create programs. Programming features such as looping and conditional execution are also created while editing samples of actual data, making them easier to comprehend. We demonstrate several uses for Dome, including web harvesting, web page creation and visual programming.

Certificate of Originality

I hereby declare that this submission is my own work and that, to the best of my knowledge and belief, it contains no material previously published or written by another person nor material which to a substantial extent has been accepted for the award of any other degree or diploma of a university or other institute of higher learning, except where due acknowledgement is made in the text.

I also declare that the intellectual content of this thesis is the product of my own work, even though I may have received assistance from others on style, presentation and language expression.

Thomas Leonard

Contents

List of Figures

Acknowledgements

I would like to thank my supervisor, Hugh Glaser, for his enthusiasm for the project, and for his feedback on this thesis.

I would also like to thank Nick Gibbins, Nigel Shadbolt and Steve Harris for providing the opportunity to test Dome as part of a large web harvesting exercise, and our three undergraduate students, Christopher Dawes, Paul Kitchin, and Isaac Norman, who provided a great deal of useful feedback after using Dome.

David Faure, Alex Larsson, Seth Nickell, Keith Packard, Filip Van Raemdonck, Jonathan Blandford and Christos Zoulas provided useful feedback with the MIME specification.

Chapter 1

Markup and structured documents

For data to be used effectively, it needs to be structured and annotated to make it amenable to machine processing. XML, the Web Consortium's new markup language, promises to allow this, and vast quantities of data are now available in the format.

There is a strong need for tools that provide flexible ways to create and manipulate these documents. Just as the usefulness of having scientific data available in computer-readable CSV (*Comma Separated Values*) format is is greatly enhanced by the existence of the spreadsheet, so this new XML-based data requires a new general purpose end-user processing tool. XML is a tree-structured format, and we propose in this thesis the *tree-sheet*: a general purpose end-user tool for the manipulation of tree structured data.

We begin the thesis by describing several markup languages, and in particular XML. We will see that XML is used to create web pages, represent facts, store word processor documents and spreadsheets and control communication, among other tasks. We will also describe related technologies and tools, including those for checking the correctness of XML, converting between different XML based formats, adding presentation information to raw data and querying XML files.

Having seen the many uses of XML, we find that the tools for manipulating it are complex and beyond the reach of many users. We present Dome, our end-user XML tool, and demonstrate it being used to perform various tasks. We will show how Dome allows the user to perform complex operations by demonstrating a desired operation on some sample data, and then letting Dome process the rest of the data automatically in the same way.

With the wealth of different formats in which data is now available, it is necessary to be able to determine the type of any piece of information so that the correct method to process it can be chosen. We consider various schemes for doing this, and present our own unifying method, using XML to ensure that it is extensible and usable by a wide range of systems.

1.1 The need to organise data

Computers have seen dramatic increases in memory and storage capacity in recent years. A typical desktop PC today has more RAM than a thousand home computers of twenty years ago put together, and there have been similar changes in long term storage capacities. With this has come the need to organise and structure the information they store so that we can make effective use of it.

Early computers had very small memories, so they could only store a few thousand letters or numbers at most. Each program could only work on a single type of data, so that there was no need to record the type, and little space to do so even if it was desired. Any structuring of the data within the file was implicit; the first line of a word processor document might be the date it was created and the second line the number of pages, but only the application which created the file would know this.

As the capacity and flexibility of the systems improved, new types of data began to appear, such as images and videos. Programs had to be able to combine data of different types and from different sources, which required each piece of data to be annotated with its type. Standardised formats had to be created to allow, for example, a desktop publishing package to be able to import an image from a painting program.

The Internet has made increasing quantities of information available to everyone. Using this resource effectively requires efficient searching, processing and machine understanding of this wealth of data. Since machines do not currently have the intelligence to understand natural languages, documents must be structured and annotated so that automatic processing becomes feasible and reliable.

In the future, there is hope that machines will be able not only to recognise formats which they already understand, but to learn how to understand new formats from formal descriptions, called *ontologies*. Thus, a program which only understands dimensions in metres will be able to make use of a file it discovers which stores them in inches, provided it can find another document to relate the two concepts. Likewise, a search for a ‘person’ with a particular name will still be able to find a match for someone described as a ‘student’ rather than as a ‘person’, by using an ontology to discover that all students are people.

1.2 A ubiquitous format for annotating data

The first step towards this goal is to adopt a general purpose method of annotating data. A key new technology in this domain is XML, the eXtensible Markup Language. XML can be used to say that a particular number is the *price* of something, or that some piece of text is the *title* of the document.

This markup can be applied to larger units of data too; XML can be used to indicate that a section of a book is a *chapter* (which may itself contain regions of text which are titles or prices, of course). XML documents therefore have a tree structure, with a single root element, such as 'book', containing branches, such as 'chapter', and sub-branches which can be nested to any depth. For example, a 'section' may contain a 'paragraph', the text of which may itself contain a 'quote'.

Meta-data is *data about data*. The author, title and copyright holder of a document are all examples of meta-data. XML can be used to mark up both data and meta-data, and to indicate which is which. Items of meta-data can often be processed independently of the data they annotate. A word-processor does not need to understand the meta-data element recording which email the document came from, and the email program does not need to understand the document it annotates.

> "The possible uses of the Web seem endless, but there the technology is missing a crucial piece. Missing is a part of the Web which contains *information about information* — labelling, cataloguing and descriptive information structured in such a way that allows Web pages to be properly searched and processed in particular *by computer*. In other words, what is now very much needed on the Web is *metadata*."
>
> — W3C Metadata Activity Statement[145]

With this additional information available, we are able to process information automatically in many interesting ways, provided we have appropriate tools. For example, we can instruct a computer to find the cheapest supplier of some item, or to index all our documents by title.

Though the technical abilities of XML are only an incremental improvement over formats that have gone before, XML has been vastly more successful politically. It is quickly becoming the format of choice for most information on the Internet, with many sub-formats designed for the different types of data. It is XML's wide-spread adoption and support that make it so important.

1.3 End-user programming tools

Programmers are already well-provided with tools to perform tasks on XML documents, but non-programmers are not so fortunate. The core of this thesis concerns a tool called Dome, which aims to bring XML processing within the reach of non-programmers by taking a visual approach to handling it, and thus presenting users with a more familiar style of interface. Because XML is a tree-structured format, we call such a tool a *tree-sheet*, by analogy to the spreadsheet.

The spreadsheet is a highly successful tool for allowing users to manipulate tables of numerical data, but its usefulness in other areas (such as the processing of web pages) is limited, and no equivalent tool exists. With the growing popularity of XML and the web, there is a strong need for end-user tools supporting general purpose processing of tree structured data.

Dome combines a familiar direct-manipulation style user interface with the flexibility of a simple programming language, as with macros in a spreadsheet. A 'record' mode is used to create programs by editing an example of the data to be processed, and programming features such as looping and conditional execution are also created while editing samples of actual data, making them easier to comprehend.

1.4 Structure of this thesis

In chapter 2 we will take a much more detailed look at the core of XML, expanding this in chapter 3 to a wide range of related technologies, and in particular those relating to the Internet. This will give a firm basis for understanding the rest of the thesis.

In chapter 4 we will look at the requirements for end-user XML processing tools and introduce Dome, an implementation of a tree-sheet. We will show how Dome can be used for such tasks as reformatting documents, and will explain the evolution of its design.

Chapter 5 uses web harvesting as a case-study for Dome. We show how Dome can be used to collect information from web sites and convert it to a more useful form for querying.

General transformation of XML documents is covered in chapter 6, where we compare XSLT (introduced in chapter 2) with Dome from a user interface point of view. We show that XSLT transformations can be automatically converted into Dome programs.

Dome is a visual language, and in chapter 7 we examine this aspect in more detail, describing previous visual languages which have influenced Dome's development, and showing that Dome is useful for other tasks besides simply processing XML.

Chapter 8 looks at a related aspect of the web — MIME types. A new XML-based format for storing and sharing MIME information is proposed to unify the many existing Unix systems in use today.

Conclusions and a summary of the work can be found in chapter 9.

Chapter 2

XML — the Extensible Markup Language

In chapter 1 we found a strong need to structure and annotate information. There have been many markup languages in the past, but none have achieved the popularity, and therefore usefulness, of XML. In this chapter, we will give a detailed description of XML's syntax and the closely related technologies of validation and querying. In the following chapter, we will broaden this to examine XML's current and future uses as a core part of the web's infrastructure.

The prevalence of XML is the motivation for our XML manipulation tool, introduced in chapter 4. This chapter will provide the technical information useful to understanding how and why XML is being used in so many ways, and will introduce concepts needed to understand the rest of the thesis. The information provided here is also useful in its own right to anyone wishing to make use of XML.

XML has become extremely popular, due to its simplicity and flexibility, and to the wide range of supporting technologies being developed for it. A huge number of applications, including word processors, vector drawing packages and databases now use XML-based formats for storage, and many communication protocols are using XML to integrate heterogeneous systems. XML is also becoming widely used for storing configuration information, as in the MIME database described in chapter 8.

2.1 Related markup languages

There have been many markup languages with similar aims to those of XML. XML's great achievement is political as much as technical, in that it has become popular enough that many different systems use it as a common format. This section gives some background information about several predecessors of XML, but is not central to the thesis.

2.1.1 SGML

XML is heavily based on *SGML*[66] (the *Standard Generalized Markup Language*). However, SGML is rather complex and cumbersome; XML is a simplified subset which is easier to implement and which has enjoyed much greater success. The SGML specification is not available on-line, which might also have slowed adoption, although other resources such as *A Gentle Introduction to SGML*[126] make up for this somewhat. A list of differences between SGML and XML can be found in *Comparison of SGML and XML*[35]. Most of the information given in this chapter about the syntax of XML also applies to SGML.

SGML was developed in the early 1980s as a standardised and extended form of IBM's *GML* system, created in 1969. The progress from the origins of GML to the completion of SGML are chronicled in *A Brief History of the Development of SGML*[68].

2.1.2 OEM

The *Object Exchange Model*[55, 104] (*OEM*) was developed by the database community circa 1995 to provide a standard format for integrating databases and transferring information online.

An OEM object is represented by a label and a value, together enclosed in <> characters. The value can be either an atomic type (anything from an integer to a video clip; the key point is that the model treats it as opaque) or a complex type, which contains a list of other OEM objects. Here, *Birthday* and *Day* are labels and `{...}` and `7` are the corresponding values:

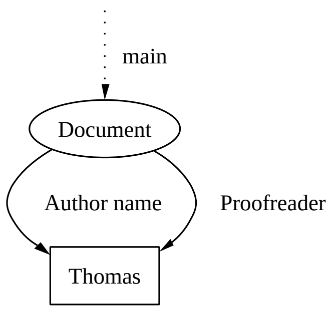
```
<Birthday {
<Month "January">
<Day 7>
<Year 1972>
}>
```

A type may be specified before the value, and any object may be labelled with a Symbolic Object Identifier (*SymOid*), if it is to be referred to from elsewhere in the document. If the SymOid is followed by two colons rather than one, then it is visible from outside of the document. The & symbol can be used to refer to an object by its SymOid.

Syntactically, OEM borrows heavily from the C programming language, from its string quoting rules to its comments and markers. Labels can be quoted if non-identifier characters need to be used. Here is a more complicated example:

```
/* This object may be referred to as &n1: */
<n1: "Author name" str "Thomas">  // The ‘str’ is optional here
/* This object can be referenced from outside the document */
<main:: Document { <&n1> <Proofreader &n1> }>
```

OEM allows implementations to use either a model where the labels are on the objects, or one where they are on the arcs linking objects. If the first type of model is used, then link labels such as ‘Proofreader’ in the example above are ignored, while if the second is used then the default name for an arc is taken from the object, but may be overridden by specifying a new name explicitly, as with our ‘Proofreader’ example, which might be parsed into a graph like this:



OEM does not allow per-object namespaces. Rather, the namespace for all object labels and types is considered to be the source of the OEM file. If OEM files from different systems are combined then the merging process will rename them as appropriate (and must, therefore, understand what they mean). For example, if one library uses ‘year’ to mean the year a book was written and another uses it to record the year it was published, then the combined file might use the labels ‘year-written’ and ‘year-published’ instead.

2.1.3 S-Expressions

S-Expressions (*Symbolic Expressions*) were created for the *LISP*[90] (*LISt Processor*) language. An S-Expression is defined to be either an atomic value, or a pair of simpler S-Expressions, written as $(S_1 \cdot S_2)$. A list of S-Expressions is represented by either a special symbol, *NIL*, for the empty list, or by a pair containing the value of the first list element, and the rest of the list.

For simplicity, this is written using the short-hand notation of a list of space separated S-Expressions enclosed in parentheses:

$$\begin{aligned} NIL &= () \\ (S_1 \cdot (S_2 \cdot (S_3 \cdot NIL))) &= (S_1 \quad S_2 \quad S_3) \end{aligned}$$

Thus, the birthday example given in section 2.1.2 could be written as:

$$(Birthday \quad (Month\ January) \quad (Day\ 7) \quad (Year\ 1972))$$

That is, the first item in the list is the name of the element and the remaining items are its content. It can be difficult to work with large S-Expressions since there tends to be a long string of consecutive closing parenthesis at the end of each one.

> "As for S-Expressions, I can see the arguments, and can't honestly tell you why the same technologists who ignored decades of S-Expression lore instantly took up XML. It's crystal-clear that you could have used S-Expression syntax for XML and it all would have worked about as well.
> Maybe it's because S-Expressions were too closely identified with the tattered dreams of the AI community? Or maybe just because XML's compulsory end-tags make it a little easier to read?"
>
> — Tim Bray (one of the creators of XML)[21]

2.2 XML core technologies

The following sections provide a more detailed look at some of the core aspects of XML. Of these, an understanding of XML's syntax, namespaces, DTDs and the XPath language are most important to the rest of this thesis.

2.2.1 Syntax

An XML document is a textual representation of an abstract data type, called an *XML Information Set*[139]. This is modelled as a tree whose root, corresponding to the XML document as a whole, is the *document information item*. Figures 2.1 and 2.2 show a diagram of an information set and the corresponding XML file.

An XML document starts with the *XML declaration*, which gives the version of XML being followed and the character set encoding of the document. This declaration is optional, but strongly recommended.

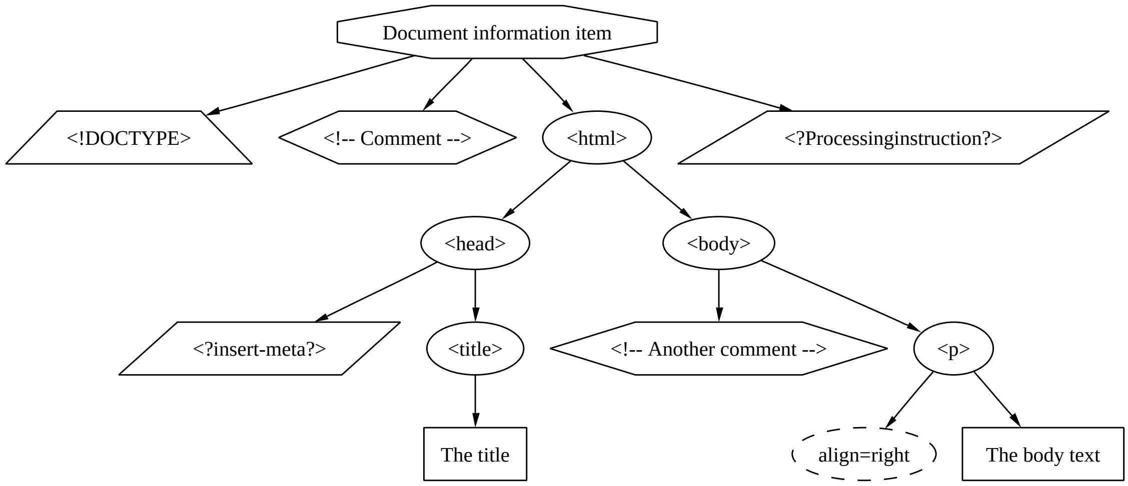


Figure 2.1: An example XML Information Set.

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE ...>
<!-- Comment -->
<html>
  <head>
    <?insert-meta?>
    <title>The title</title>
  </head>
  <body>
    <!-- Another comment -->
    <p align="right">
      The body text.
    </p>
  </body>
</html>
<?ProcessingInstruction?>
```

Figure 2.2: The XML file corresponding to figure 2.1.

The optional *document type declaration* (DTD) may be given next. The use of this is explained in section 2.2.3.

The document contains a tree of *elements*. Each element has a *start tag*, such as `<title>`, and an *end tag*, such as `</title>`. These must nest correctly, with each element having a matching pair of tags. Between an element's opening and closing tags is the element's content; its child nodes in sequence. The root of this element tree (`<html>` in the example) is called the *document element node*, and it is a child of the document information item. Unlike OEM, XML disallows certain characters, such as spaces, from being used in element names.

The *attribute nodes* of an element node are written as a list of `name="value"` pairs in the element's opening tag. The order is unimportant, and there cannot be two attributes with the same name for a single element.

An element which has nothing between its opening and closing tags can be written using the short-hand notation `<tag/>`, which is equivalent to `<tag></tag>`. In this thesis, the notation `<element>` will be used for both tags (the textual markers in the XML file that surround an element's content) and the elements themselves; it will be clear from the context which is meant.

The OEM birthday example from section 2.1.2 can be written in XML in a similar form, except for the additional closing tags. It could also be written using the shorter but less extensible attribute notation instead (note that XML treats these forms as different; it is up to the designer of an XML-based language to decide which form should be used):

```
<Birthday>
  <Month>January</Month>
  <Day>7</Day>
  <Year>1972</Year>
</Birthday>

<Birthday Month="January" Day="7" Year="1972"/>
```

Comments and processing instructions may appear at any point in the document after the declaration. Comments are for human readers of the raw XML and normally have no effect on the use of the document. Processing instructions can be used to change the way the document is processed, and are application-specific. Neither are particularly important to the content of this thesis.

2.2.2 Namespaces

Since XML allows anyone to define their own element names, there is considerable risk that the same name will be used to mean different things. As long as documents are kept separate this is not a problem, but if an application is expected to handle a document of any type, or if two documents must be merged, such conflicts must be resolved.

In OEM, the namespace issue is solved by treating element names from different document creators as being in separate namespaces. However, this requires knowing where a document came from, or finding a way to store the (globally unique) name of the creator in the document. When merging documents from different sources, the merging application must understand the meanings of elements from all the source documents and then define its own set of element names which covers all the input documents. This means that an operation as seemingly simple as embedding an object from some application into a web page requires the web page composer application to have knowledge of every possible source application, and requires each web browser to have knowledge of every possible composer. This is clearly impractical.

XML solves the problem using *XML Namespaces*[20]. The meaning of each attribute and element is uniquely identified by a $(namespaceURI, localName)$ pair. The *namespace URI* can be any globally unique string, and authors can ensure that it *is* globally unique by choosing a URI which they control, taking advantage of the globally unique nature of the Internet's *DNS* system[94].

Thus, the `<head>` element from traditional HTML is identified, in the new XHTML, by the pair (http://www.w3.org/1999/xhtml, *head*). Since URIs are rather long to type, and contain characters which aren't allowed in XML names, the namespace is bound to a *prefix* and referred to later using the prefix to stand for the full URI.

When using this system, ':' becomes a special character in element and attribute names, being used to separate the $prefix$ from the $localName$. A prefix is bound to a namespace using a special '`xmlns:prefix`' attribute, and that prefix applies to the element and all its descendants recursively, unless one of them rebinds it to something else, rather as lexically-scoped variables work in many programming languages. The *default namespace* (the namespace of elements with no prefix) can also be bound in a similar way. In this example, the `<html>` and `<p>` elements are in the XHTML namespace, while the `<mathml:math>` element identifies the `<math>` element from the MathML namespace:

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:mathml="http://www.w3.org/1998/Math/MathML">
  ...
  <p>Here's an interesting formula:</p>
  <mathml:math>...</mathml:math>
  ...
</html>
```

It is worth noting that the default namespace applies only to elements, not to attributes. Attributes with no prefix have no namespace (they are not affected by the default namespace setting). Attributes which have no namespace must be read in the context of their parent element node. An attribute is typically only namespaced if its meaning is independent of its element, such as the '`xlink:href`' attribute, described in section 3.3.3, which can be used to link any XML element to another resource.

The choice of prefix is unimportant as far as readers of the XML document are concerned, since the XML parser expands the prefixes into full namespace URIs, and the application matches on these.

When a web browser is displaying a page it can correctly identify, for example, an embedded *MathML*[5] subtree and render it without any risk that the object is something else which just happens to use the same element name. When an XHTML composer application wishes to embed a MathML formula from some program into a web page, it can put it in directly without having to rename anything.

There are a number of difficult issues caused by namespaces, however. From a programmer's point of view, there is the difficulty of keeping the prefix bindings consistent with their usage. For example, a copy-and-paste operation that moves an `<html:p>` element may have to create a new binding for the 'html' prefix (if none is in scope in the new context), change the prefix to one which *is* in scope, or rename the prefix to an unused name.

These problems can be lessened by ignoring the prefixes entirely while processing and generating new ones when the document is written out, although this is made difficult by the fact that the DOM interface (see section 2.2.6) treats namespace bindings as attributes, as does the core XML specification, while XPath (see section 2.3.1), which is often combined with DOM, treats namespaces as a separate axis (though this is likely to be deprecated in XPath 2.0). There is also the problem of searching, which is discussed in the XPath section below.

Elements not bound to any namespace are *not* considered to be in a 'Null' or '' namespace — this would contradict the XML Namespaces requirement that a $(namespaceURI, localName)$ pair uniquely identifies an element type. Rather, an element with no prefix, and used where no default namespace has been declared, is not in any namespace.

A detailed exploration of the issues of XML namespaces can be found in *Namespace Myths Exploded*[18] and the *XML Namespaces FAQ*[17].

2.2.3 Document Type Declaration (DTD)

All XML documents must be *well-formed*. This allows them to be parsed into a tree structure. In addition, a document may be *validated* against a *DTD* to constrain the possible structure and values further. A *valid* document is one which can be validated. A *validated* document is one which has been processed with a validator, possibly being changed in the process by the insertion of default values. The `<!DOCTYPE>` declaration gives the name of the *document element* and specifies the type for every element and attribute appearing in the document.

For example, the following document declares itself to have a `<quiz>` document element which contains one or more `<problem>`s. Each of these contains a

`<question>` followed by an `<answer>`. A `<question>` contains textual data[1], while an `<answer>` contains either a `<yes>` or a `<no>`. `<yes>` and `<no>` are both defined to be empty (leaf nodes).

Finally, we allow `<problem>` to have a '`difficulty`' attribute; if missing, the default 'easy' is used. A default value may be preceded by the *#FIXED* keyword to indicate that no other value may be given. Instead of a default value, either of two keywords may be used; *#REQUIRED* indicates that a value for the attribute *must* be specified, while *#IMPLIED* indicates that the attribute is optional.

```
<?xml version='1.0'?>
<!DOCTYPE quiz [
  <!ELEMENT quiz (problem)+>
  <!ELEMENT problem (question, answer)>
  <!ELEMENT question (#PCDATA)>
  <!ELEMENT answer (yes|no)>
  <!ELEMENT yes EMPTY>
  <!ELEMENT no EMPTY>
  <!ATTLIST problem
   difficulty CDATA "easy">
]>
<quiz>
  <problem difficulty='very easy'>
    <question>1+1 = 2?</question>
    <answer><yes/></answer>
  </problem>
  <problem>
    <question>2+2 = 1?</question>
    <answer><no/></answer>
  </problem>
</quiz>
```

The above *content model* is said to be *local* since it is defined within the document itself. More commonly, the DTD is stored in a separate document. The exact syntax and full range of features of the `<!DOCTYPE>` is defined in the XML specification[140].

2.2.4 Entities

Entities are constants which may be declared in the DTD and referred to later. A validating parser will perform macro expansion on them when the XML is loaded.

[1]PCDATA is a historical term meaning "parsed character data". SGML allowed a DTD to specify the content as 'CDATA' instead, so that certain characters within it would have their special meaning ignored, but in XML no element can be declared to contain CDATA (and no attribute can contain PCDATA).

An *internal entity* is declared with its replacement text. This text is parsed as a *document fragment* (a series of complete nodes). Referring to the entity at any point in the document is equivalent to typing the replacement text directly.

An *external entity* is specified using the `SYSTEM` keyword. In this case, the replacement text is read from the named resource[2]. An external entity definition may also include a *public identifier*, which allows an XML parser to fetch well-known external entities from an alternate location.

The following DTD extract defines two entities, one internal and one external:

```
<!ENTITY intEnt "<emph>Replacement text</emph>">
<!ENTITY extEnt SYSTEM "external.xml">
```

They may be referred to later by surrounding the entities' names with `&` and `;`, as follows:

```
<p>&intEnt; is the internal element's replacement text.</p>
<p>&extEnt; is the contents of the external.xml file.</p>
```

Parameter entities perform a similar function to normal entities, but are expanded within the DTD rather than the main document (allowing a DTD to include definitions from another DTD, for example). These start with a `%` rather than an `&`.

An *unparsed entity* is an external entity which is specially marked by the `NDATA` keyword followed by a notation declared in the DTD with a `NOTATION` declaration; this is intended to help an application understand how to handle the resource. Such an entity can be used only as the value for an attribute declared as type `ENTITY`, and is used to refer to an external binary resource (such as a PNG image file). They have little to do with the other types of entity, and are rarely used.

2.2.5 Schema languages

While DTDs, examined in section 2.2.3, are very simple and provide a good starting point for checking document correctness, they lack flexibility.

The main problems with DTDs are:

- DTDs do not support namespaces. This is one of the most important features of XML, and means that DTDs are only really useful for checking locally generated documents which use exactly the same prefixes as the DTD. Externally generated documents using different prefixes, which mean the same thing in XML, would be rejected by the DTD.

[2]The named resource is similar to an XML file, except that the XML declaration at the start *must* include the encoding, and the rest is a document fragment (the usual rule about having only a single top-level element node does not apply).

- The syntax for defining DTDs is not extensible; DTDs do not consist of element and text nodes, but have their own syntax entirely. This means that changes and extensions to the DTD language require changes to the core XML grammar, which is supposed to be very stable. Also, it is not possible to include one XML document as a subtree in another and keep both DTDs.
- The DTD language is not expressive enough to make a number of useful constraints on the content, such as requiring an attribute's value to be a positive integer, or allowing the same element to have different content depending upon the context in which it appears.

Other schema languages, such as those described below, can be used to perform more complex checks on the document. While the DTD and these other schemata are entirely independent, the document is normally validated against the DTD first so that entity references are expanded.

In *DSD: A Schema Language for XML*[74], the authors of the DSD schema language suggest that *"A useful schema notation must: identify most of the syntactic requirements that the documents in the user domain follow; allow efficient parsing; be readable to the user; allow limited tree transformations corresponding to the insertion of defaults; be modular and extensible to support evolving classes of XML documents"*. However, not everyone would agree with the need for a schema to insert defaults; James Clark argues, in *RELAX NG and W3C XML Schema*[71], that *not* having the document modified by the validation process is an advantage of his RELAX NG system.

XML Schema

XML Schema[138] is the official XML schema language of the W3C, and is therefore likely to enjoy wide-spread support. Like all the schema languages examined here, it uses XML for its own syntax. The specification is split into two parts, the first defining the language and syntax and the second defining a large selection of built-in datatypes. This second part has proven more popular than the first, with several other schema languages, and other systems, choosing to use it for their own type systems.

There are two groups of types in XML Schema. A *complex type* is one which has child elements or attributes, while a *simple type* is one which doesn't. Elements can be of either type, but an attribute must always have a simple type.

As well as including a large number of predefined types, the language provides many ways to create new ones, by deriving from other types. The fragment at the start of figure 2.3 defines a new type called 'question', which is derived from the 'xsd:string' type (types are namespaced), but restricted to matching the given

```
<?xml version='1.0'?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name='question'>
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\d+ *[-+*/]\d+ *= *\d+\?"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:complexType name='yesno'>
    <xsd:choice>
      <xsd:element name='yes'><xsd:complexType/></xsd:element>
      <xsd:element name='no'><xsd:complexType/></xsd:element>
    </xsd:choice>
  </xsd:complexType>

  <xsd:element name='quiz'>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name='problem' maxOccurs='unbounded'>
          <xsd:complexType>
            <xsd:attribute name='difficulty' type='xsd:string' default='easy'/>
            <xsd:sequence>
              <xsd:element name='question' type='question'/>
              <xsd:element name='answer' type='yesno'>
              </xsd:element>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Figure 2.3: An XML Schema for the quiz example.

regular expression[3]. The aspects of a simple type that can be restricted are called its *facets*. New simple types can also be defined as *lists* (space separated sequences) or *unions* of other simple types.

An XML Schema contains a number of *global elements*, one of which must match the *document element* of the document being validated. Each global element has a '`name`' (to match the element name in the document) and a type. The type may be one defined elsewhere in the document, or even outside of it, or it may be an *anonymous type*, defined within the global element itself. If the type is missing or 'anyType' then the element is said to have *open content*; ie, anything can appear inside it, something that cannot be specified using a DTD.

[3]The example expression requires a question to be of the form '$A\langle op\rangle B = C?$'. '\d+' means one or more digits, ' *' means zero or more spaces, and '[-+*/]' means any of the four listed operators.

The '`minOccurs`' and '`maxOccurs`' attributes can be used to indicate how many times an element may appear. Both default to 1. However, a different syntax is used for attributes; the '`use`' attribute is set to 'optional', 'required' or 'prohibited'.

Default values also work differently for attributes and elements. A default value for an attribute is used if the attribute is not given, whereas the default value for an element is used if the element is present but empty.

Elements can be combined in various ways. `<sequence>` indicates that each of its child elements must appear (subject to their '`minOccurs`' setting) in the given order; `<all>` requires each of the elements listed within it to appear exactly once, but in any order; and `<choice>` allows only one of its elements to appear. Again, attributes are treated differently, and XML Schema does not allow a choice between attributes.

`<group>` allows a selection of elements to be inserted, rather like *parameter entities* in a DTD. `<attributeGroup>` performs a similar function for attributes.

As can be seen from figure 2.3, XML Schema has a rather verbose and unwieldy syntax. There are many other well-known problems with XML schema, such as those noted in *Problems with XML Schema*[98].

RELAX NG

RELAX NG[34] is a joint effort by the authors of its two predecessors, TREX and RELAX. It aims to be a schema language that is *"simple, easy to learn, uses XML syntax, does not change the information set of an XML document, supports XML namespaces, treats attributes uniformly with elements so far as possible, has unrestricted support for unordered content, has unrestricted support for mixed content, has a solid theoretical basis, and can partner with a separate datatyping language"*. It is on the way to becoming an ISO standard.

Figure 2.4 shows a possible schema for our quiz example. The `<choice>` operator works like `|` in the DTD and the elements `<oneOrMore>` and `<zeroOrMore>` are equivalent to `+` and `*`. The schema is equivalent to the DTD given above, except that no default value is given for the '`difficulty`' attribute. This is because RELAX NG is purely a validator; the content of the document does not change depending upon whether it was validated or not.

Notice that the structure of the RELAX NG schema follows that of the format it describes. It also allows parts of the schema to be defined in one place and reused, allowing a syntax more similar to the DTD, if required. An interesting feature of RELAX NG is that it treats elements and attributes in a very similar way allowing such restrictions as "this element must have either a 'name' attribute or a 'name' child node, but not both".

```
<?xml version='1.0'?>
<element name='quiz' xmlns="http://relaxng.org/ns/structure/1.0">
  <oneOrMore>
    <element name='problem'>
      <attribute name='difficulty'/>
      <element name='question'><text/></element>
      <element name='answer'>
        <choice>
          <element name='yes'><empty/></element>
          <element name='no'><empty/></element>
        </choice>
      </element>
    </element>
  </oneOrMore>
</element>
```

Figure 2.4: A RELAX NG Schema for the quiz example.

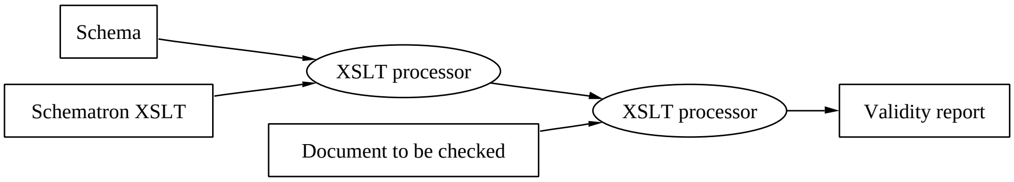


Figure 2.5: Using the XSLT implementation of Schematron.

Datatypes are interpreted by the validator, not specified by the specification. This allows RELAX NG to use datatypes defined elsewhere, including the ones specified in the XML Schema specification, part II, as in this example from the RELAX NG tutorial[33]:

```
<element name="point"
         datatypeLibrary="http://www.w3.org/2001/XMLSchema-datatypes">
  <element name="x">
    <data type="double"/>
  </element>
  <element name="y">
    <data type="double"/>
  </element>
</element>
```

Schematron

Schematron[45] is an assertion based schema language. A Schematron schema gives a list of checks to perform. Schematron itself is available as an XSLT transformation which turns a Schematron schema into an XSLT transform which can, in turn, be used to validate documents (see figure 2.5). It also uses XPath expressions (discussed in section 2.3.1) for the tests, and so is extremely portable.

The schema below enforces the first few rules from our DTD (the rest have been omitted to save space). Notice that in Schematron rules are defined to spot *invalid* documents, rather than to declare what a valid document looks like. This means that Schematron schemata are more likely to miss certain checks, though the range of checks than can be performed is much larger than with the grammar-based schema languages. Of course, it is quite possible to use another system for most of the checks on the document, and just use Schematron for the few tests that are otherwise impossible to specify.

```
<?xml version='1.0'?>
<schema xmlns="http://www.ascc.net/xml/schematron">
  <pattern>
    <rule context="/">
      <assert test="quiz">Top-level element must be 'quiz'</assert>
    </rule>
    <rule context="/quiz">
      <assert test="count(*) = count(problem)">
        Only problem elements may be in quiz.
      </assert>
      <assert test="count(problem) > 0">
        There must be at least one problem in a quiz.
      </assert>
    </rule>
  </pattern>
  ...
</schema>
```

Schematron is also unusual in that the message to display for each problem is included in the schema, giving more friendly reports. There are a selection of Schematron XSLT implementations available to cater for different output styles. The most basic simply print the messages contained in the `<assert>` elements on failure, while the more sophisticated generate HTML pages which link the errors to parts of the source document.

Other schema languages

There are many other schema languages available. In Dave Raggett's *Assertion Grammars*[109], a list of assertions are made, similar to those in a DTD. The assertions are slightly more flexible, and may be made in any order, making it easier to create modular systems.

The *Document Structure Description*[74] (DSD) language is a more ambitious replacement for XML Schema. Unlike XML Schema, DSD is powerful enough to fully describe itself. It is able to easily express constraints such as that an `<a>` anchor node in xhtml may not contain another `<a>`. It defines primitive datatypes using regular expressions, though it does not use the traditional syntax, but rather reuses the constructs used for specifying element content. DSD also has an interesting approach to default insertion: as well as specifying defaults in the schema, subtrees in the document can specify defaults by inlining a DSD fragment, allowing these defaults to take effect even in the absence of a schema. In its current state DSD does not support namespaces, and so is of limited practical use.

Other schema languages include *XML-Data*[79], which aimed to provide an XML syntax for DTDs and extend it with information useful to relational databases, *XML-Data Reduced*[51] (*XDR*), a simplified version of XML-Data, and three older submissions to the W3C: *Schema for Object-oriented XML*[42] (*SOX*), *Document Definition Markup Language*[16] (*DDML*), and *Document Content Description for XML*[19] (*DCD*).

A comparison of the schema languages DTD, XML Schema, XDR, SOX, Schematron and DSD can be found in *Comparative analysis of six XML schema languages*[80]. Schematron was found to be the most expressive, although the grammar based approach of XML Schema and others was thought to be a better approach in general.

2.2.6 Parsing and serialising

An advantage of using a standard markup language is that programs can use standard parsers. Traditionally, programmers have created a new parser for every new format, leading to the same bugs appearing over and over again (such as failure to correctly escape special characters, support non-English alphabets, or to detect and report errors). XML promises to solve this once and for all, by allowing all the low level parsing to be done by a standard XML parsing library.

There are two main approaches taken by such libraries. The event based model, as used by *SAX*[91], the *Simple API for XML*, scans the source document in order, invoking methods in the application whenever it has parsed a feature of the input. For example, a SAX parser will read in an entire start tag (such as `<p align="right">`) and invoke the 'startElement' method with the element's name and namespace and a list of its attributes, all correctly unescaped. Event based parsers use very little memory, since they only have to maintain the current stack of parent elements, and are most useful for single-pass operations on the data, such as searching.

By contrast, tree based systems parse the entire document into a tree structure in memory, allowing the application to do much more complicated querying and manipulation. By far the best known of these is the Web Consortium's *DOM*[141] (the *Document Object Model*).

```
n = 1
for chapter in doc.documentElement.getElementsByTagName('chapter'):
    for title in chapter.getElementsByTagName('title'):
        print "Chapter %d: %s" % (n, title.childNodes[0].data)
        n += 1
```

Figure 2.6: Creating a numbered list of contents with a DOM model.

```
n = 1
filter_elements(parser, None).next()
for chapter in filter_elements(parser, 'chapter'):
    for title in filter_elements(parser, 'title'):
        print "Chapter %d: %s" % (n, parser.getEvent()[1].data)
        n += 1
        parser.expandNode(title)
```

Figure 2.7: XMLPull code equivalent to figure 2.6.

Tree based systems are also useful when the manipulated data must be serialised back to XML. Although people often try to create XML output manually (using `print` statements, or similar) this often leads to broken XML. Uche Ogbuji gives a good summary of the traps for the unwary in *Proper XML Output in Python*[103].

DOM-type systems are usually much simpler to program for, because the programming language's normal control flow can be used while processing. SAX's callback based approach, by contrast, requires storing all state information explicitly across calls. However, the large data structures created by DOM parsers mean that they are typically much slower. A possible solution to this is the *XMLPull* interface[122, 125], which delivers events using an iterator model (the program requests the next event) rather than a callback one.

To compare these methods, consider the problem of creating a table of contents by printing and numbering `<title>` elements, but only those directly within a `<chapter>` element (this structure is found in *DocBook*[96] documents, for example). It is not necessary to understand the example code below in detail; it is enough to note the similarity in size and structure between the methods.

Figure 2.6 shows a simple DOM-based system (written in *Python*[131]). The XMLPull version of this code, shown in figure 2.7, is very similar in structure. The short `filter_elements` function is not part of PullXML, but is sufficiently generic that it would probably be a worth-while addition to the core XMLPull API. The function definition is given in appendix A, section A.1.

By contrast, a SAX based approach requires us to define startElement(), endElement() and characters() methods, and to create a structure which tracks the current parser's depth and whether the second level element was a `<chapter>` and the third a `<title>`. This code, which is considerably more involved than the DOM and XML-Pull examples given above, can be found in appendix A, section A.2.

In *What's Wrong with XML APIs*[58], two further styles of XML parsing are identified: the *data-binding API* and the *query API*. A 'query API' simply takes a query in another language, such as XPath, and returns the result. A 'data-binding API' uses a schema to present the data as ordinary classes and objects. For example, if the schema describes a `<person>` element which has an attribute '`age`' whose type is 'xsi:integer' then the API will present this as a class 'person' which has an integer attribute 'age'.

There are many XML libraries now available to programmers, such as GNOME's *libxml*[54], Apache's *Xerces*[4], James Clark's *expat*[37], and Fourthought's *4Suite*[50], to list just a few of the free ones.

2.2.7 History

The earliest XML specification[136] available from the W3C is dated 14-Nov-1996, and tells us that:

> "XML was developed by a Generic SGML Editorial Review Board formed under the auspices of the W3 Consortium in 1996 and chaired by Jon Bosak of Sun Microsystems, with the very active participation of a Generic SGML Working Group also organized by the W3C."

The document lists a number of goals, simplicity being a running theme. XML documents form a subset of the possible SGML[66] documents, so that any XML document can be read by an SGML parser, and it will produce the same parse tree (for a comparison between the two, see *Comparison of SGML and XML*[35]).

The specification has changed little since then. Apart from minor changes (such as the use of `apos` and `quot` instead of `sq` and `dq` for single and double quotes, and handling of whitespace), the original specification required support for *undistinguished empty elements*. In these, `<tag>` is allowed to mean `<tag/>`, if declared as such in the DTD. Thankfully, this quirk (intended for better SGML compatibility) was removed, allowing XML's useful feature of being able to parse any document into a tree without the DTD (originally, a document could be *valid* without being *well-formed*).

Some special processing instructions could also be used at the start as a simpler alternative to giving a full DTD. This was probably intended to allow non-validating parsers to cope with the undistinguished empty elements.

Since its 1.0 release in 1998, the XML standard has barely changed, except for some minor corrections. The new version 1.1[38] (previously known as *Blueberry*[39]) will only update Unicode character support to bring XML in line with the current *Unicode specification*[130], and deal with some other minor issues relating to character encodings.

In general, XML is a very stable platform upon which many other technologies are being built, although, as some of the original designers explain:

> "Of course, success brought pressures to fix the things the Working Group got wrong. Experience working with XML has shown that parts of the design don't work quite as well as other parts. The mechanisms for declaring and using entities, the rules for processing well-formed documents, and the limited possibilities of nesting full XML documents inside other XML documents are all occasionally sources of difficulty."
>
> — Hollander and Sperberg-McQueen[61]

2.3 Querying and transformation

In this section we examine how XML documents can be processed. XPath is the core component here, allowing evaluation of expressions referring to values within an XML document. XSLT is a transformation language for turning one XML document into another, typically by adding styling and formatting information to it. XQuery builds upon XPath to produce query results which are themselves XML. While both XSLT and XQuery technically do the same job, they have different goals and correspondingly different syntax.

2.3.1 XPath

It is often necessary to identify a node or set of nodes within an XML document. *XPath*[144] provides a flexible syntax for this. It is used in many other standards, including XPointer, XSLT and XQuery (described elsewhere in this chapter).

XPath expressions normally evaluate to a set of nodes, but they can also be used to return other values, such as strings and integers. For example, `1+1` is an XPath expression that evaluates to the integer 2.

XPaths can be quite complex, but in the simplest case they look like file pathnames; `/` evaluates to the *document node* (the document itself, not the toplevel element inside it), and `/html` to the *document element node* (assuming the toplevel element is `<html>`).

Since an XML element may contain multiple child nodes with the same name, even a simple expression may evaluate to multiple nodes, so `/html/body/table` is the set of `<table>` nodes which are children of the `<body>` child of the `<html>` toplevel element.

By default, each step of the XPath selects from the list of *child nodes*. This is called the *child axis*, and the above can be written more verbosely as `/child::html/child::body/child::table`. Other axes allow selecting nodes with other relationships to the nodes selected by the previous step. For example, the *parent axis* contains only the context node's parent node, the *attribute axis* (usually abbreviated to `@`) contains the attributes of the context element, and the *descendant-or-self* axis (`//`) contains all nodes within the context node's subtree.

A sub-expression in brackets may be used to filter the matching nodes. Most commonly, this is just a number and selects only the nth match (eg, `/html/body/table[1]` to select the first table node only). However, the predicate may itself be an XPath expression. The expression `//*[@uri]` selects every element in the document which has a '`uri`' attribute. The `*` matches any element node, and the XPath `@uri` is evaluated in the context of each one, returning either an empty node-set (which is treated as 'false') or one containing the attribute, which is treated as 'true'.

Relative XPath expressions are ones which do not start with a `/`. These are evaluated relative to some *context node* passed as input to the evaluator. For example, if the first cell of the first row of an HTML table is the context node, then evaluating the relative XPath `../following-sibling::tr[1]/child::td[1]` will return the first cell of the next row after this node's parent [4].

One major difficulty with XPath is its interaction with namespaces. Since XPath interprets an element with no prefix as having no namespace, while DOM treats such an element as being in the default namespace, it is not possible to write such search queries in the same form in both places.

Also, searches must be namespace aware if they are to work across different documents, so namespace bindings must be set up for the search expression itself. These bindings cannot be inferred from context, since there may be several possibilities. Consider a user who enters the XPath `/html/table` as a search expression. The document may look like this:

```
<html>
  <table xmlns='http://example.com'/>
  <table xmlns='http://test.org'/>
</html>
```

In this case, it is not clear what the user is searching for unless they state the namespaces explicitly, which is rather cumbersome, since the two `<table>` elements have different namespaces (and, therefore, different meanings). The user must

[4] XPath defines a number of shorthand notations; `..` selects the single node on the parent axis and so works much like `..` in a pathname.

search for 'html/example:table', and then explicitly bind the new 'example' prefix to the appropriate namespace. In section 4.5.1 we show how our XML editor handles this problem.

2.3.2 Xcerpt

Xcerpt[24] provides an alternative match expression syntax to XPath. In this, patterns are specified as trees themselves. The notation given is textual, and relies on four different types of bracket to indicate whether an element's child nodes must appear in order ([ ]) or not ({}) and whether additional child nodes may be present (indicated by double brackets). For example, this expression from the paper will match an element a which exactly contains the child nodes b, c and g, in that order. c must contain child nodes d and e, although the order is not important and other nodes may also be present:

$$a[b, c\{\{d, e\}\}, g]$$

The corresponding XPath is rather involved:

```
a[*[1][self::b] and *[2][self::c[d and e]]
        and *[3][self::g] and count(*) = 3]
```

Xcerpt allows variables to be bound to parts of the matched subtree (using $\leadsto$), allowing functional programming style pattern matching. The *desc* keyword indicates that the match may be at any depth:

$$a[X_1 \leadsto \mathrm{desc} f[c, d], X_2, e[\,]]$$

The above pattern matches an element a containing, in order, three elements. The first must either be an element f, with child nodes c and d in order, or contain one as a descendant. This part of the match is assigned to the X_1 variable. The second can be any element, and is bound to X_2. The third must be a e element with no child nodes.

Xcerpt also has a query language which can be used to build a result document using Xcerpt queries. This is similar to XQuery (below), but using a logic-programming style. The Xcerpt tutorial[119] provides an introduction to this language.

2.3.3 XSLT

XSLT[36] is a language for transforming XML documents. It was originally part of *XSL*[1] (the *Extensible Stylesheet Language*), which adds formatting information to documents. For example, a typical transformation might replace each `<author>`

```
<xsl:template match='/'>
  <html>
    <body>
      <h1>Table of Contents</h1>
      <ol>
        <xsl:apply-templates select='/book/chapter'/>
      </ol>
    </body>
  </html>
</xsl:template>

<xsl:template match='chapter'>
 <li><xsl:value-of select='title'/></li>
</xsl:template>
```

Figure 2.8: XSLT to transform a DocBook document into an HTML table of contents.

element in the source document with a `<fo:block font-weight="bold">` element in the transform's output. XSL was later split into *XSL Formatting Objects*, providing a vocabulary for expressing document formatting, and *XSL Transformations* (XSLT), a general purpose transformation language.

An XSLT transformation is a set of *templates*. Each template matches a particular type of node in the source document, and contains some content to create in the output document. When a stylesheet is applied to a source document, the XSLT processor finds the template which matches the document node and *instantiates* it to create the result document.

Instantiating the template may involve creating literal output nodes, calculating values (using XPath) and recursively instantiating more templates, typically to handle the child nodes of the source node being processed. XSLT will be examined in more detail in chapter 6. For now, the sample XSLT in figure 2.8, which creates an HTML contents page by listing all the chapter titles in a DocBook[96] document, should indicate the general syntax.

2.3.4 XQuery

XQuery[137] (the *XML Query* language), extends XPath in a number of ways. Any XPath expression is a valid XQuery, but XQuery has additional syntax to create new elements using XML tags. When constructing such elements, values enclosed in `{}` braces are evaluated as XQuery subexpressions rather than as literal text. For example, '`1+1`' returns the number 2, while '`<answer>{1+1}</answer>`' returns `<answer>2</answer>`.

The intended uses of XQuery are set out in detail in the *XML query use cases*[30] document, but the most common use is to combine information from records in multiple XML documents into a query result, providing an XML equivalent to the SQL language used in relational databases.

XQuery provides an equivalent to *list comprehensions* from functional programming[63], with its *FLWR* (For, Let, Where, Return) syntax, as shown in this example from *XQuery: An XML query language*[31]:

```
for $i in document("items.xml")/*/item
    let $b := document("bids.xml")/*/bid[itemno=$i/itemno]
        where count ($b) > 10
            return <popular-item> {
                        $i/itemno,
                        $i/description,
                        <bid-count>{count($b)}</bid-count>
                      }
                   </popular-item>
```

This returns a sequence of `<popular-item>` elements, one for each item with more than ten bids, combining information from two source documents. In a FLWR expression the 'where' filter clause is optional, and either of the 'for' or 'let' parts may be omitted, but not both.

An XQuery consists of a single expression, the *query body*, such as the one above, and an optional *prolog*, which contains function definitions, module imports and otherwise sets up the environment for processing the query.

XQuery is still only a working draft, although it is expected to become an official recommendation of the W3C soon. Because of this, implementations tend to implement slightly different revisions of the specification at present. Also under development is *XQueryX*, an (extremely verbose) XML syntax for XQuery, intended for machine transformation.

Some other XML query languages, *LOREL*, *XML-QL*, *XML-GL*, *XSL* and *XQL*, are compared in *Comparative Analysis of Five XML Query Languages*[13].

2.4 Summary

In this chapter we have taken a brief look at the history of XML and its predecessors, described in detail the syntax of XML, including namespaces and DTDs, and examined validation, querying and transformation of XML documents.

We have seen that the distinction between query and transformation languages is only in their intended use. A query is a transformation which transforms the document to be searched into the search result. While we might expect a query to produce a simple plain text result (such as a number), and a transformation to produce output in XML, in fact XML-Query typically produces an XML document as the result of the query, whilst XSLT is quite capable of producing plain text output, or output in some other format such as nroff[5].

Indeed, even validation can be handled by a transformation language. Schemata can be implemented as XSLT transforms which use the language's error reporting facilities to signal incorrect data, although it would also be possible to produce a boolean result instead. And, of course, some 'validation' systems, including XML Schema, really do modify the document (by adding default attributes, for example).

In the next chapter we will look at some of the practical uses of XML, and in particular the ways it is being used in various Internet-related technologies. Having seen some of the uses of XML and the need for powerful methods to manipulate it, we will introduce Dome, our user-friendly programmable XML editor, in chapter 4.

[5]This author uses DocBook XML for the reference manual of some of his software, and uses an XSLT transformation to create an nroff format Unix man-page from one of the appendixes, while a second stylesheet creates HTML of the whole document.

Chapter 3

XML and the Internet

In chapter 2 we looked at XML and some of the core technologies surrounding it. In this chapter, we broaden our scope to see how XML is being used to support the infrastructure of the Internet. We will look at shared XML-based formats, which are allowing document interchange between different applications, the various ways that XML is used for two-way communication between programs over the network, and the rôle of XML in the future of the Web. This will provide a strong motivation for end user XML processing tools, such as our Dome editor, introduced in chapter 4.

In section 3.1 we examine the way XML is being used to store documents, such as spreadsheets, artwork and texts. A programmable XML editor will allow the user to perform operations such as indexing their work, recovering data from a document for which they do not have a dedicated editor program, or performing repetitive work automatically when their regular software lacks programmability.

When sending documents across the network, XML is used not just to represent the data being sent, but is often involved in controlling the transmission itself. In section 3.2 we investigate means of communicating in or with XML; such XML-based communication formats allow Dome programs to communicate with other systems, as we shall see in chapter 5. We continue the network theme in section 3.3, where methods for locating and referring to documents on the Internet are described.

Section 3.4 combines the areas of data representation and networking with a look at the *Semantic Web*, in which XML is used to provide information in a way that is suitable for processing by computers, not just for display to humans. The Semantic Web seeks to find a general way to represent knowledge, so that information from any number of sources can be combined and processed automatically. User tools are required here to help users add the required semantic markup to their own documents, and to use the semantic information provided by others.

3.1 Storing documents in XML

Many programs are now using XML based data formats. Since XML handles all the low-level parsing, escaping and structuring, XML formats are typically easier to import and export. Because they are largely plain-text, and therefore human-readable, people feel more confident about archiving important information in XML-based formats, since it is clear that the information could be recovered later even if the original software was lost or would not run on new systems. Another advantage is that both users and software authors can easily check the conformance of a program by validating the documents it produces using one of the many XML schema languages available (as described in section 2.2.5).

There have been several successful XML-based formats already; some examples include *DocBook*[96] (for technical documentation), *XHTML*[132] (for web pages), *XBEL*[46] (the *XML Bookmark Exchange Language*), and *SVG*[114] (*Scalable Vector Graphics*). A particularly important format still under development is the OASIS standards group's XML format for office documents; their 'must' requirements are that the format:

- ... be suitable for office documents containing text, spreadsheets, charts, and graphical documents.
- ... be compatible with the W3C Extensible Markup Language (XML) v1.0 and W3C Namespaces in XML v1.0 specifications.
- ... retain high-level information suitable for editing the document.
- ... be friendly to transformations using XSLT *or similar XML-based languages or tools.*

The last point is particularly significant; the format is explicitly intended to allow general purpose tools, such as Dome, to process the documents. When documents in this and other XML-based formats are combined with general-purpose XML tools, many interesting operations become available, such as extracting all the chapter headings in a word processor document using XPath, or creating a hypertext link to a section within a document. These tasks can be scripted and automated.

The tree-structure of XML makes it easy to embed one XML document within another; for example, an XHTML web page could in-line an SVG image directly, and some *JavaScript*[99] within the page could then manipulate the page and the image together, perhaps highlighting the part of the diagram corresponding to the text the pointer is over, even though the JavaScript language itself knows nothing of the SVG format.

3.2 Communicating with XML

Since structured data formats are ideal for connecting independent systems (possibly following different versions of various standards), XML is a popular choice for new message formats. In this section, we will be concerned not with the data being transferred, but with the way the communication itself is controlled.

3.2.1 BEEP — frame based communication

BEEP[112] (the *Blocks Extensible Exchange Protocol*) provides a light-weight framework for two-way communication streams. A client connects to a server, and each sends a *greeting frame* to the other. The client and server then send frames back and forth between them over the connection (which may be a TCP stream[113], for example).

Each frame begins with a simple textual header, a blank line, and then the body. The body is treated as binary data, whose length and type is given in the header. The type, which is a *MIME type* (see chapter 8), defaults to 'application/octet-stream' so that there is very little overhead when sending raw binary data. BEEP is suitable for efficiently sending large amounts of binary data between applications, automatically splitting large blocks of data across multiple frames.

BEEP can multiplex many channels over a single connection by giving each channel a number and quoting it in the header of each frame. Although a frame's payload need not be XML, there is a special control channel which uses an XML based format. This is the only channel available when a new BEEP session is established, and it is used to send the greeting and negotiate the creation of other channels.

3.2.2 Jabber instant messaging

Jabber[70, 118] is a communications protocol intended for *instant-messaging*. It uses XML to make the protocol extensible, and is intended to allow not only person-to-person communication, but also machine-to-machine.

The client–server protocol is based on *XML streams*. An XML stream is started by the client sending a (namespaced) <stream:stream> opening tag, at which point the server sends the same tag on the reply stream. Either party can then asynchronously send complete XML child elements over the stream; these are the units of communication. The connection is terminated by closing the top-level element with </stream:stream>. XML streams do not contain processing instructions, comments, DTDs or entities (except for the few defined by the XML standard itself).

A Jabber session begins with authentication, after which three elements can be used: `<message>`, `<iq>` (information/request) and `<presence>`. Messages are information sent from one user to another, rather like emails. Presence information is used to indicate that a client is on-line, so that other clients know that this one is available now. IQs are used to get and set various bits of information, such as finding out details about another client.

Jabber is intended to be very simple so that programs can use it to communicate between themselves, as well as to make writing jabber clients easier. Most of the complexity is moved to the servers, which store messages waiting to be delivered, and forward messages between themselves as needed to get them to their final destinations.

3.2.3 SOAP messages

The *Simple Object Access Protocol*[56] (*SOAP*) provides a standardised *envelope* for an XML payload. The envelope, which is itself XML, consists of a *header* and a *body*. The complete SOAP message is then sent using an existing transport mechanism, such as *HTTP*, *SMTP* (email), or *BEEP*.

The header contains zero or more *blocks* containing data intended for the systems (called *actors*) which will handle the message on the way to its destination. Each *header block* is addressed to an actor, and has application specific contents such as authorisation, routing information and caching hints. Header blocks with no actor, and all *body blocks*, are for the *anonymous actor*; the message's final destination.

The SOAP specification also details the *SOAP Encoding Style*, which is a way to serialise a graph of objects into XML, although applications can, if they wish, specify any other encoding system.

A particularly common type of SOAP message is the *SOAP RPC* (*Remote Procedure Call*). Here, the body contains an element which is the name of a method or function provided by the remote system, and its child elements are the method's *in* and *inout* parameters. Here is an example of a SOAP RPC call to the *ROX-Filer*[82] file manager[1], asking it to create a panel window. It

[1]ROX-Filer was written by the author, but is not a part of this thesis.

corresponds to the function call ‘`Panel(Name = "Default", Side = "Bottom")`’.

```
<?xml version="1.0"?>
<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-envelope">
 <env:Body xmlns="http://rox.sourceforge.net/SOAP/ROX-Filer">
  <Panel>
   <Name>Default</Name>
   <Side>Bottom</Side>
  </Panel>
 </env:Body>
</env:Envelope>
```

A SOAP message is considered to be one-way, but may generate a reply message containing the method's return value and the value of each *out* and *inout* parameter.

SOAP messages may be sent using various different transports. The SOAP/HTTP transport defines a method to send SOAP messages using the HTTP protocol. The sample message above would normally be sent using the *XSOAP* protocol (devised by the author, primarily for ROX-Filer), which uses the *X Window System*[146] as its transport.

SOAP RPC is being used by several companies, including Google and Amazon, to provide *web services* — program-friendly interfaces for accessing services normally only available using HTML (typically via a web browser). A program trying to perform an action such as buying a book online would normally have to carefully parse HTML pages intended for human readers to do its job. This process is complicated and error-prone, and programs are likely to break when the site layout changes.

SOAP services provide an entirely separate interface to the site's functions. This is a different approach to that of *stylesheets* (see section 3.4.1), which attempt to make the same pages readable to both humans and computers.

The *Web Services Description Language*[32] (*WSDL*) is an interface description language for SOAP. It allows a client machine to find out what SOAP messages a server supports. WSDL documents can be registered with a *Universal Description, Discovery and Integration*[129] (*UDDI*) index, allowing users to quickly locate providers of particular types of service.

The different methods of communication described above have different strengths. BEEP is most suitable for tasks where the control information must impose only a small overhead, or when large amounts of binary data are being sent; an XML stream is simpler when there is only one channel and no binary data; and SOAP is most useful when the communication is divided into separate message–reply pairs, rather than being stream based.

3.3 References and links

It is frequently necessary to refer to a particular resource available through the Internet. In this section we look at the three parts of this process: identifying a resource, such as a web page; identifying a point or range within that resource, such as a paragraph of text; and creating the link from some source to the target.

A *resource* is generally a complete file, fetched whole over the network. A *fragment* is a part of the resource identified by the client (the program that has fetched the resource) after the resource has been retrieved. For example, when a client wishes to read a particular paragraph of text, it will fetch the whole HTML web page and then search for the paragraph itself, rather than request that only the single paragraph itself be sent.

3.3.1 Uniform Resource Identifiers

A *Uniform Resource Identifier*[10] (*URI*) is a human-readable string typically composed of a *scheme*, a *host*, and a *resource* on that host available using that scheme. Here is a sample URI, for the document describing URIs:

```
http://www.ietf.org/rfc/rfc2396.txt
```

In this URI, 'http' is the scheme, 'www.ietf.org' is the host, and '/rfc/rfc2396.txt' is the resource. Any program which understands the 'http' scheme can connect to the computer named 'www.ietf.org' and request the document using the HTTP protocol. The scheme part is present in all URIs, but the rest depends on the scheme. Many schemes follow the above syntax, however.

A *Uniform Resource Locator* (*URL*) is a URI which identifies the resource by its location (as in the example above). A *Uniform Resource Name* (*URN*) is a URI which identifies a resource by its name or by its properties, such as the *ISBN number* for a book, in the form 'urn:isbn:0854-327932'.

A URI is often followed by a *fragment identifier*, which identifies a specific part of the resource (such as a particular paragraph within an article). This is interpreted by the receiver after the resource has been obtained, and so is not considered part of the URI. A full identifier (URI and optional fragment identifier) is called a *URI Reference*.

Only a subset of the US-ASCII characters can appear in a URI. Any octet outside of the allowed range must be given in the escaped form '%HH', where 'HH' is the two-digit base 16 value of the octet. URIs do not contain any indication of the character set used in such escaping. An *Internationalized Resource Identifier*[100] (*IRI*) is a URI which, when unescaped, forms a *UTF-8*[147] encoded string. The

UTF-8 encoding is able to represent all the characters of the world in a single encoding; it is intended to replace the previous system of using one encoding for each alphabet.

D. J. Bernstein's *Internationalized domain names*[12] proposal discusses some of the security and compatibility issues involved in supporting non-ASCII characters in domain names, which are often found in URIs.

3.3.2 Selecting fragments within resources

XPointer[142] provides a general way to reference parts of a document, by allowing a number of different addressing schemes to be used, one of which is an extended version of *XPath* (described in section 2.3.1). An XPointer is normally combined with a URI (see section 3.3.1) to identify a part of the named document.

In the simplest case, an XPointer expression is the value of an ID attribute in the document, and identifies that single node; this is very similar to an HTML fragment identifier.

Otherwise, an XPointer is a list of $(SchemeName, Data)$ pairs. An XPointer processor tries each pair in turn until one returns a non-empty set of nodes. The main scheme is, rather confusingly, also called 'xpointer'. It is based on XPath, but extends it to include *points* and *ranges*, in addition to the usual *nodes*. Ranges, points and nodes are all *locations*.

A point is defined as "A location in an XML Information Set with no content or children", such as the point between two adjacent nodes or characters. A $(ParentNode, Index)$ pair can be used to identify a point. For text or attribute parent nodes, the index counts characters into the textual content, otherwise the index counts child nodes.

A range is "An identification of all the XML Information Set content between a pair of points"; ie, a $(StartPoint, EndPoint)$ pair. This example, from the specification, selects everything bounded by `<REVST/>` and `<REVEND/>` (revision start and revision end) markers, including the markers themselves:

$$xpointer(descendant :: \mathrm{REVST}/\textit{range-to}(following :: \mathrm{REVEND}[1]))$$

There is a separate *xmlns scheme*, which never selects any nodes but instead is used to bind namespaces for use in other schemes.

3.3.3 Linking resources together

XLink[143] (the *XML Linking Language*) defines a method to link different resources together. An *arc* connects two resources in one direction. The most

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE sample [
  <!ELEMENT sample (a)*>
  <!ELEMENT a (#PCDATA)>
  <!ATTLIST sample xmlns:xlink CDATA #FIXED "http://www.w3.org/1999/xlink">
  <!ATTLIST a      xlink:type CDATA #FIXED "simple"
                   xlink:href CDATA #REQUIRED>
]>
<sample>
 <a xlink:href="resource.xml">Some resource</a>
</sample>
```

Figure 3.1: Using a DTD, it is possible to make elements similar to HTML's `<a>` element.

common type of link is one embedded in a document (such as a web page) which links to some target document, such as the familiar `<a>` element from HTML. These are called *outbound arcs*. XLink also provides for *inbound arcs*, where the link is specified at the destination, and *third-party arcs*, which can be used to link two resources without embedding the link in either.

XLink defines attributes, rather than elements, for linking, allowing any element in an XML document to function as a link. These attributes must be in the XLink namespace (namespaced attributes are called *global attributes*; recall that attributes are not normally in any namespace and are recognised instead by the namespace and name of their parent element). Any element used in XLink must have an '`xlink:type`' attribute. This normally has the value 'simple' for an `<a>`-style outbound arc from the element itself to a remote resource. A DTD may specify a default value for this attribute, for example, to declare every `<a>` element to have a fixed `xlink:type='simple'` attribute — see figure 3.1.

Sometimes it is not possible to specify a link in the source document, either because the document is read-only, or because the resource is not XML. In that case, a *linkbase*, a document containing a list of 'extended' type links between other resources, can be used. An element with an `xlink:type='extended'` attribute is an extended link. It contains a list of resource *locators* (each naming a resource that participates in the link) and a list of arcs joining these resources.

3.4 The Semantic Web

> "The Semantic Web is an extension of the current web in which information is given well-defined meaning, better enabling computers and people to work in cooperation."
>
> — Tim Berners-Lee, James Hendler, Ora Lassila[11]

The primary goal of the *Semantic Web* is to provide data in a form that machines can process easily. By ‘process’, we mean performing tasks such as indexing documents by category, comparing the prices for some item between web sites, or making logical inferences based on the information, rather than merely displaying it for a human to read. One approach to this is to provide the same information twice, once in *HTML*[135] and once in some computer-readable format. However, it is clear from experience that maintaining the same information twice is too much effort.

From early simple meta-data systems, such as indicating whether a page is suitable for children using *PICS*[133] (*Platform for Internet Content Selection*), more complex systems such as RDF have emerged.

3.4.1 XML and stylesheets

One solution is to store the information in a computer readable format, and provide some kind of *transformation* (called a *stylesheet*) which will allow this to be formatted for display. Consider the following XML TODO list:

```
<?xml version='1.0'?>
<?xml-stylesheet href='test-style.css' type='text/css'?>
<todo>
  <job>
    <date>20-Jan-2003</date>
    <actor>Me</actor>
    <task>Mow lawn</task>
  </job>
  <job>
    <date>21-Jan-2003</date>
    <actor>Me</actor>
    <task>Laundry</task>
  </job>
</todo>
```

It is clear that a program could be written to process this document easily, without worrying that the format will change just because the author has tired of some particular layout. All the layout information is contained within a stylesheet, such as this one:

```
todo { padding: 1cm; font: 20pt sans; }
job { display: block }
actor { color: red }
date { color: blue }
```

The above example specifies that the TODO list should be presented in a 20pt sans-serif font with a 1cm border around it. Each job will go on its own line,

```
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version='1.0'>

<xsl:template match='/todo'>
  <html>
    <title>TODO List</title>
    <body style='max-width: 40em; background-color: #aaa;'>
      <div style='border: 1px solid black; background-color: #eee; padding: 0.3cm'>
        <h2 style='text-align: center; text-decoration: underline;'>TODO List</h2>
        <dl>
          <xsl:apply-templates select='job'/>
        </dl>
      </div>
    </body>
  </html>
</xsl:template>

<xsl:template match='job'>
  <dt><xsl:value-of select='date'/></dt>
  <dd><span style='color: red'><xsl:value-of select='actor'/>:</span>
      <span style='margin-left: 0.5cm'><xsl:value-of select='task'/></span></dd>
</xsl:template>

</xsl:stylesheet>
```

Figure 3.2: An XSLT stylesheet to format our TODO list.

and the actor will be coloured red and the date blue. This gives the author considerable flexibility to change the presentation (or provide multiple styles of presentation) without affecting any programs using the data directly. In fact, we could imagine that everyone in the world could agree on a common format for TODO lists, even if they all desired different styles. The default stylesheet to use for a particular XML document is often specified with a processing instruction, such as `<?xml-stylesheet?>` in the example above.

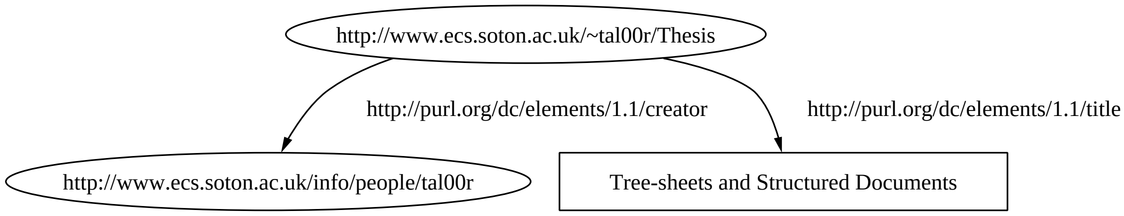
The above stylesheet is specified in *CSS*[15] (*Cascading Style Sheets*), but more powerful transformations can be specified using other languages, such as *XSLT*[36] (described briefly in section 2.3.3 and later in more detail in chapter 6).

In fact, XSLT and CSS are often combined; the sample transformation shown in figure 3.2 converts the source XML into HTML with embedded CSS for styling. It also adds extra elements, including a page title and a heading. With XSLT, the possibilities for reformatting a site while keeping to a standard fixed source format are endless.

3.4.2 The Resource Description Framework

RDF[78, 87] (The *Resource Description Framework*) provides a way to make assertions. Each assertion is a triple (*subject*, *predicate*, *object*), with each of the three parts being a globally unique identifier for that particular concept. To ensure uniqueness, URIs are used, although, as with namespace URIs, this does not mean that the named URI is a resource accessible over the web. The *object* part of the triple may also be a *literal* (such as a string) rather than a URI.

RDF assertions are usually represented as graphs, with the arcs representing the predicates. The example below gives the author and title of this thesis. The thesis is given a new unique ID so that it can be referred to again later, while the author uses an existing URI. The title is a literal value. The predicates (*creator* and *title*) are well-known concepts, identified by their well-known *Dublin Core*[47] URIs.



Using the URI of a web page to represent a person causes the slight problem that we cannot then use the web page's URI to refer to the page itself (as a distinct concept from the person it describes). However, we can use whatever URIs we please as long as we don't use the same one for two RDF concepts — we could make up a new URI to represent the web page if it became necessary. RDF attaches no importance to the choice of URI used for any particular concept, only caring whether two URIs are equal, but human creators of RDF documents often choose mnemonic identifiers.

If an RDF concept is needed within a document, but need not be referred to from outside, a *blank node* can be used. For example, if we wanted to indicate that the thesis had a completion date, and that the completion date has year, month and day arcs to literal values, we need not assign a globally unique URI to the concept of *this thesis's completion date*, but can use a blank node instead. Of course, the concept of *completion date* itself must still have an identifying URI.

The above RDF example can be written in XML as shown below. The '`rdf:about`' attribute gives the subject, while each child node of the `<rdf:Description>` describes an arc from it, named by the concatenation of the namespace URI and the local name of the element (that is, `<dc:creator>` corresponds to the RDF predicate `http://purl.org/dc/elements/1.1/creator`).

```
<?xml version="1.0"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
 xmlns:dc="http://purl.org/dc/elements/1.1/"
 xmlns:example="http://ecs.soton.ac.uk/~tal00r/example/">

  <rdf:Description rdf:about="http://www.ecs.soton.ac.uk/~tal00r/Thesis">
    <dc:creator rdf:resource="http://www.ecs.soton.ac.uk/info/people/tal00r"/>
    <dc:title>Tree-sheets and Structured Documents</dc:title>
    <example:completion>
      <rdf:Description>
        <example:day>5</example:day>
        <example:month>5</example:month>
        <example:year>2003</example:year>
      </rdf:Description>
    </example:completion>
  </rdf:Description>

</rdf:RDF>
```

The `<example:completion>` arc leads to a *blank node* (the `<rdf:Description>` has no '`rdf:about`' attribute[2]), which in turn has *day*, *month* and *year* arcs leading to literals.

Literals may be typed, again by using a URI to identify a unique type, given in the '`rdf:datatype`' attribute. XML Schema Part II (see section 2.2.5) defines a set of types, and URIs by which to refer to them. However, RDF itself does not require any particular type system to be used.

RDF objects (non-literals) may also be given a type, using an `<rdf:type>` arc to the type (class) of the object. As a helpful short-hand, if another element is used in place of the `<rdf:Description>`, then that element is considered to give the type of the object.

The above might all sound rather complicated, but with these short-cuts in mind we can rewrite our TODO list example above in RDF as follows:

```
<?xml version='1.0'?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns="http://www.ecs.soton.ac.uk/~tal00r/TodoList/">
  <todo rdf:ID="main">
    <job rdf:parseType="Resource">
      <date>20-Jan-2003</date>
      <actor>Me</actor>
      <task>Mow lawn</task>
```

[2] We can, however, use an `rdf:nodeID` attribute to allow the blank node to be referred to from elsewhere in this file, but not from outside.

```
    </job>
    <job rdf:parseType="Resource">
      <date>21-Jan-2003</date>
      <actor>Me</actor>
      <task>Laundry</task>
    </job>
  </todo>
</rdf:RDF>
```

All we had to do was:

- Enclose the document element `<todo>` in an `<rdf:RDF>` element (with the appropriate namespaces).
- Give the toplevel `<todo>` element an ID so that it can be referred to from outside the file.
- Give each `<job>` element an '`rdf:parseType`' of `Resource` (this is equivalent to enclosing the child nodes of each `<job>` in an `<rdf:Resource>`) to create a blank node for each one.

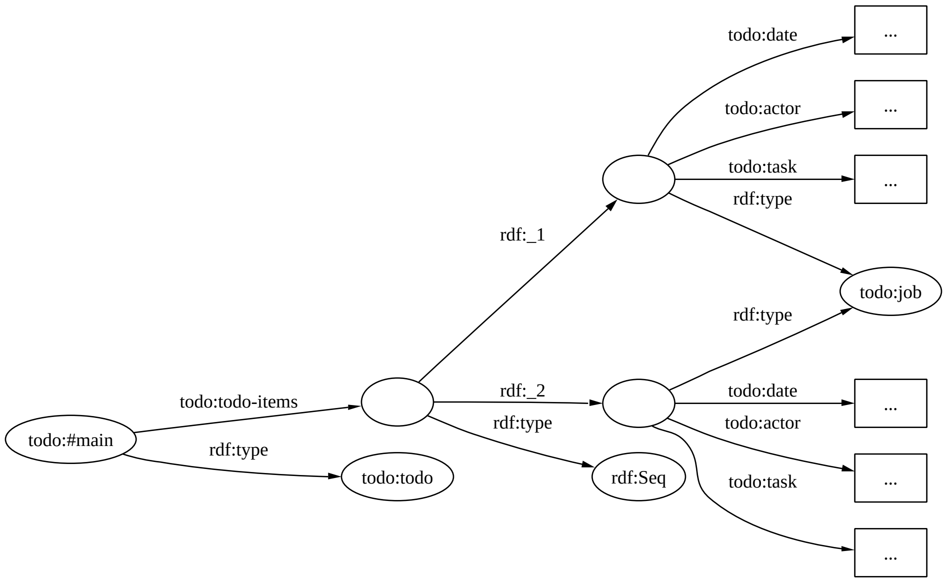
Since the structure of the new document is so similar to the old one, it is easy to write and maintain it, and adjust our stylesheets slightly to cope with it. To make our new document look even more like the previous one, we could fix the '`rdf:parseType`' and '`rdf:ID`' attribute values in the DTD, rather than specifying them in every file.

When an RDF-aware program reads our file, the ordering information in the XML is lost, but that wasn't significant in this example anyway. If it had been, we could have used an `<rdf:Seq>` *container*. RDF provides three types of container: `<rdf:Bag>`, `<rdf:Seq>` and `<rdf:Alt>`. A Bag is an unordered collection of resources, a Seq is an ordered sequence, and an Alt indicates a list of alternatives (for example, the same resource at different locations or in different languages). When used, each of these containers is a blank node, whose type is one of the three given above, and the arcs from the blank node are named `<rdf:_1>`, `<rdf:_2>` and so on. Another short-hand when writing RDF in XML is to use the `<rdf:li>` element in place of a number to number the elements automatically:

```
<?xml version='1.0'?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:todo="http://www.ecs.soton.ac.uk/~tal00r/TodoList/">
  <todo:todo rdf:ID="main">
    <todo:todo-items>
      <rdf:Seq>
        <rdf:li>
          <todo:job>
```

```
            <todo:date>20-Jan-2003</todo:date>
            <todo:actor>Me</todo:actor>
            <todo:task>Mow lawn</todo:task>
          </todo:job>
        </rdf:li>
        <rdf:li>
          <todo:job>
            <todo:date>21-Jan-2003</todo:date>
            <todo:actor>Me</todo:actor>
            <todo:task>Laundry</todo:task>
          </todo:job>
        </rdf:li>
      </rdf:Seq>
    </todo:todo-items>
  </todo:todo>
</rdf:RDF>
```

The above XML corresponds to the following graph representation:



3.4.3 Uses of RDF

This section gives two examples applications where RDF is being used already. RDF is also used internally in various systems; the Mozilla browser in particular stores large amounts of metadata in RDF, including the list of downloaded files and recently entered URLs. While applications for RDF are still somewhat thin on the ground, it is being strongly pushed by the Web Consortium and is expected to become widely used.

Many sites provide lists of new items or products. To make these machine readable, the *RSS*[108] format can be used. There are actually two versions of the format;

the older, simpler one (*Really Simple Syndication*) and a newer RDF-based one (*RDF Site Summary*).

This common format allows portals and user tools to collect information from multiple sources and present it to the user all at once. The use of *Dublin Core*[47] ensures that RSS browsers can recognise the different elements correctly. RSS is one of the most popular uses of RDF at present.

Annotea[75] is an annotation system for the web. It uses RDF to associate a document, or part of a document, with an annotation, allowing web-page readers to annotate pages which they don't own. Different groups of people can run their own Annotea servers so that comments are only shared within particular groups (such as 'teachers' and 'students'), or annotations can be private to a single user, perhaps to attach some memorandum to a web page for when the page is revisited.

3.4.4 RDF Schema

RDF Schema[22] (*RDFS*) provides additional *RDF vocabulary* which can be used to provide a more powerful type system for RDF, such as supporting inheritance. RDFS defines the `rdfs:Class` resource, and any other RDF resource can have an `<rdf:type>` arc to this to indicate that it is a class. Having declared something to be a class, instances of the class set their `<rdf:type>` to the class's URI. A class S can have an `<rdfs:subClassOf>` giving a parent class P. Any object which has an `<rdf:type>` as S is then automatically considered to also have an `<rdf:type>` of P.

Properties (other arcs) can be defined in a similar way; each property is declared as a resource having a type of `rdf:Property`. The property is then used as the predicate in another triple. Each property declaration has an `<rdfs:range>` giving the class of which the property's value must be an instance, and an `<rdfs:domain>` likewise restricting the subject's class. Multiple ranges and domains may be specified for a property; the intersection is used. The range may also be a literal type instead of a class. `<rdfs:subPropertyOf>` works analogously to `<rdfs:subClassOf>` allowing, for example, *hasSibling* to be declared as a sub-property of *hasRelative*.

The following set of example triples may make this all a little clearer; it declares 'hasParent' to be a property which relates two people, and declares Mary's parent to be Bob:

example:hasParent	rdf:type	rdf:Property
example:hasParent	rdf:domain	example:Person
example:hasParent	rdf:range	example:Person
example:Mary	example:hasParent	example:Bob

3.4.5 Ontology languages

An *ontology* is a formal representation of some knowledge domain. It defines a number of terms, and expresses relationships between them. RDF Schema provides some limited possibilities for describing an ontology.

SHOE[59] (*Simple HTML Ontology Extensions*) was a system allowing web page authors to mark up their pages (either using a text editor or the provided tool) to provide more semantic information about the content. A web-crawler called *Exposé* indexed the information, allowing very accurate search results for marked-up pages.

The query interface would ask the user to select an ontology (from a list of every ontology known to the search engine) and then to choose a class, such as *employee*. The user is then offered a list of relations that apply to that class. Relations from parent classes (such as *age* in the *person* class) are not shown, since the user should have just selected the parent class in the first place, but *inverse relations* (where another ontology has a relation whose second argument is this one) are shown. This means that users don't have to choose between the *company* and *employee* classes depending on whether the relation is expressed as *works-for* or *has-employee*. Using SHOE, a user could search for all employees of a particular company and get no irrelevant results.

SHOE was later superseded by *DAML+OIL* (a combination of the *DARPA Agent Markup Language*[44] and the *Ontology Inference Layer*[48]), which then evolved into *OWL*[124] (the *Web Ontology Language*). OWL extends RDF and RDF Schema with more expressive power. The *Use Cases and Requirements* document[60] points out that "*RDF Schema cannot specify that the Person and Car classes are disjoint, or that a string quartet has exactly four musicians as members*". The language provides constructs to relate different ontologies, such as converting measurements between different units, or splitting a date string into its component fields.

OWL Lite provides vocabulary to indicate that two *classes*, *properties* or *individuals* (objects) are simply different names for the same thing. This allows information from multiple ontologies to be combined even if they use different URIs for the same concept. OWL Lite can also be used to indicate that two individuals are different and can express extra information about properties, such as the fact that *hasParent* and *hasChild* are inverses, or that *supersedes* is transitive.

OWL DL extends OWL Lite, for example by allowing the statement that a *Person* cannot be an instance of both the *Man* and *Woman* classes, while still allowing queries to be solved efficiently. *OWL Full* extends OWL DL even further, but cannot guarantee that all queries will be solvable in finite time. The OWL language is currently still being finalised, and is the focus of many research projects.

3.5 Conclusions

We have seen that XML has become extremely popular, and is forming the basis for many new specifications, allowing considerable code reuse and easing interoperability between systems.

Several popular methods of communication using XML have been described: BEEP for light-weight multiplexed channels controlled by XML messages; XML streams for long running communications involving lots of small XML fragments; Jabber, which uses XML streams, to provide instant message delivery between people or systems; and SOAP for one-shot messages. SOAP RPC, used to make method calls on remote objects, will be discussed further in chapter 5, where we show how Dome can access web services using it.

Whether SOAP will prove a popular way to provide web services is, however, somewhat unclear. A stylesheet based approach using a powerful transformation language such as XSLT brings the same advantages of allowing programs to easily process information, but without the need to maintain two separate systems. However, SOAP RPC *is* being used in some places as a replacement for inter-process communication systems such as *CORBA*[102], where there is no human interface at all. SOAP is not as fast or lightweight as CORBA, but it is much easier to use, and is popular in places where processing speed is not critical.

Background information about referencing and linking resources on the web was given, largely as an aid to understanding the World Wide Web, and specifically the Semantic Web. We have seen how web pages can be designed to provide information that is useful to both human readers and to automatic processing systems, using either stylesheets or RDF, and how ontology languages aim to describe and connect the different vocabularies used with RDF around the world.

The Semantic Web is a major focus of current research, with most real-world adoption only in the area of stylesheets (principally CSS, but moving to XSLT as browser support becomes more widespread). However, the popularity of stylesheets is mainly due to the increased opportunities for attractive layouts, and the ability to quickly re-theme a site, rather than due to a desire to make machine processing easier.

There is less drive for semantically rich web pages, although RSS has been made popular by a number of news aggregation services. *August 2009: How Google beat Amazon and Ebay to the Semantic Web*[49] describes a future in which everyone uses search engines to intelligently find goods, so that site owners are motivated to design their sites to be as easy to index as possible.

In the next chapter we tackle the problem of how to let users take advantage of all these XML-based technologies. While domain-specific applications may hide the details from users, there is a clear niche for an end-user tool to perform more general manipulation of XML documents. This is the goal for Dome, our programmable, visual XML editor.

Chapter 4

Dome: An XML processor

From our tour of the world of XML conducted in chapters 2 and 3 we can see that vast amounts of data are available, from all over the Internet, to anyone with the appropriate tools to process them. In this chapter we introduce *Dome*, our own general purpose XML-processing application.

We have already seen some of the other XML tools available: the XPath expression language allows us to extract values from an XML document; XSLT provides powerful transformation abilities, often used to add styling information; XQuery provides an alternative to XSLT more focused towards performing calculations on the data; and we have seen several schema languages, used to check the validity of XML documents.

However, these tools are not, on their own at least, appropriate for the majority of computer users. At the other end of the spectrum, we have specialised editors which hide the XML completely. An example of this type of program is *Sodipodi* (an *SVG* image editor), which provides a familiar direct manipulation drawing interface to the user. These programs are certainly easily usable, but they lack the flexibility to cope with arbitrary XML documents.

The *spreadsheet* is the most successful visual language to date. It allows non-programmers to process tables of numerical data in a highly flexible way. By displaying the data being edited on screen, and by letting the user see the results of any calculations they request as they write them, the spreadsheet brings many complex tasks within their reach.

Just as a spreadsheet is useful for almost any task involving tables of data, so a general purpose XML editor will be useful for a wide range of XML-related tasks. Indeed, it will be useful for operations on any tree-structured data, since XML imposes no special limitations, and this also encompasses some simpler datatypes, such as lists, which can be thought of as being merely shallow trees. We will call such programs *treesheets*, by analogy to spreadsheets, and our target audience will be those with abilities similar to those of a typical spreadsheet user.

We begin this chapter by considering the requirements for our tool. We will then describe our implementation, along with some examples of its use, contrasting it with other XML editors. The use of our tool to collect information from the Web is explained in detail in chapter 5, and we compare Dome with XSLT in chapter 6. A detailed discussion of the tool as a visual language can be found in chapter 7 where, as with spreadsheets, we find ways of using treesheets to write programs not focused on the processing of external data, but which use the tree model for their internal data structures.

4.1 Requirements

A treesheet is a general purpose editor for tree-structured data, such as XML. It must allow flexible editing of XML documents, as a text editor does for text files, or as a spreadsheet does for tables of data. Our most basic requirements, then, are that the tool must be able to load any well-formed XML document, and allow it to be changed into any other well-formed XML document. The editing operations provided must be suited to trees; an action such as 'delete subtree' is useful, whereas 'delete line' is not (since another XML processor must be able to insert or remove linebreaks in some places, for example between attributes, without changing the meaning of the data).

Below, we present some typical use cases for an XML treesheet. Of course, many of these requirements could be satisfied by writing a special purpose tool for them. The aim of a general purpose editor is to be adaptable to new situations, so the listed tasks are only intended as specific examples, not as an exhaustive list.

Viewing and manually editing XML

At its simplest, a treesheet can be useful even without its programming features, simply to view, search through, or manually edit an XML document. The view must show the tree structure clearly and provide navigation suitable for trees, such as 'move to parent node'.

The ability to display a document in *outline mode*, where the contents of certain subtrees are hidden, is useful to see the overall layout of a document. For example, displaying a word processor document in outline mode might show only chapter and section nodes, as in a table of contents, but would allow the user to expand any one of them as desired to see more detail.

Format conversion

A major goal of XML is to facilitate document interchange between different systems. A general purpose XML tool should allow conversions between XML formats. For example, a user could create a program to convert accounting information from an old package so that it could be imported by a replacement, even if the replacement didn't support the old format directly.

Creating web pages

XHTML, the language used for new web pages, is an XML-based format, and this allows such tasks as generating a summary page for the web from a database. A file listing employees and their offices could thus be converted periodically to HTML and placed on the web. The treesheet must be sufficiently programmable that the conversion only needs to be specified for one person and can process the rest of the file automatically, and that the whole task can be periodically repeated to update the page.

Web harvesting

XHTML pages can also be read by an XML treesheet, to collect information from web sites. HTML, the SGML-based mark-up language used for older web pages, can be automatically converted to XHTML; this is especially useful for recovering information from older pages. The ability to follow links between web pages so as to collect information from multiple pages in a single run would greatly extend the usefulness of a treesheet in this area.

Querying an XML file

A treesheet could be used as a replacement for *XQuery* (see section 2.3.4). A typical task would be to search an inventory list and locate all out-of-date stock, perhaps returning the result as an order request for new stock.

Reformatting

Treesheets allow complex reformatting operations on documents. For example, a *description list* (a list defining a number of terms, such as this list of use cases) could be turned into a table with the terms in the first column and the descriptions in the second.

Using web services

Web services provide a programmable interface to the features of certain web sites. A XML treesheet would allow non-programmers to take advantage of these services. A user could create a program to query several servers for the price of a book, and summarise the results.

The above tasks require an XML-aware tool, as pure text-based processing is very sensitive to slight structural changes, or to non-significant reformatting made by other XML tools working on the data. XML's quoting and escaping rules also need to be taken into account.

Consider the example task above of putting employee addresses on a company intranet web page. If the source file is plain text, then any special characters would need to be escaped. Only the most carefully crafted text-based script would correctly deal with the source line '`Bob:Rooms 1234 & 1235`'. The rest would copy the '`&`' literally to the XHTML output, instead of escaping it as '`&`' (see section 2.2.4 for information about XML entities). An XML treesheet knows about the XML escaping rules and will handle this automatically, even if the source is plain text. If the source is also XML, which would make handling of multi-line addresses much simpler, for example, then the benefits of a treesheet are even greater.

Our treesheet, Dome, is able to perform all of the above tasks. In the following sections we introduce its syntax and methods of operation. It will become clear that the tool is flexible enough to manage a wide range of tasks. The web harvesting task has been the most explored use of Dome, and this is detailed fully in chapter 5 rather than in this chapter.

4.2 Dome concepts

Dome may be used simply as an editor, and supports the familiar editor operations such as cut, copy, and paste. Once the editing of documents using these direct manipulation operations is mastered, the user may switch recording on to generate 'Dome programs'. This is analogous to the way a spreadsheet user may record macros to edit their data. In both cases, the full state of the data being worked on is shown to the user at all times.

Dome loads and saves XML format files, displaying them as a tree. The display is easier to read than a textual presentation, since the relationships between nodes are shown by their position on screen, and a single element node is represented by a single graphical element, rather than by two textual tags marking its boundary. Dome also allows any subtree to be collapsed into a single node for display purposes, making the overall structure of the file clearer by hiding unnecessary details.

Unlimited Undo and Redo buffers are provided, to allow users to experiment freely without losing their data due to mistakes. Program execution can also be undone step by step in this way; this is a useful debugging aid. Dome is also able to take advantage of the tree structure to make Undo more useful; editing can be restricted to a single subtree for while, and when the editing session is over, all the edits on that subtree are combined into a single edit at the next level up and can be undone and redone in one operation.

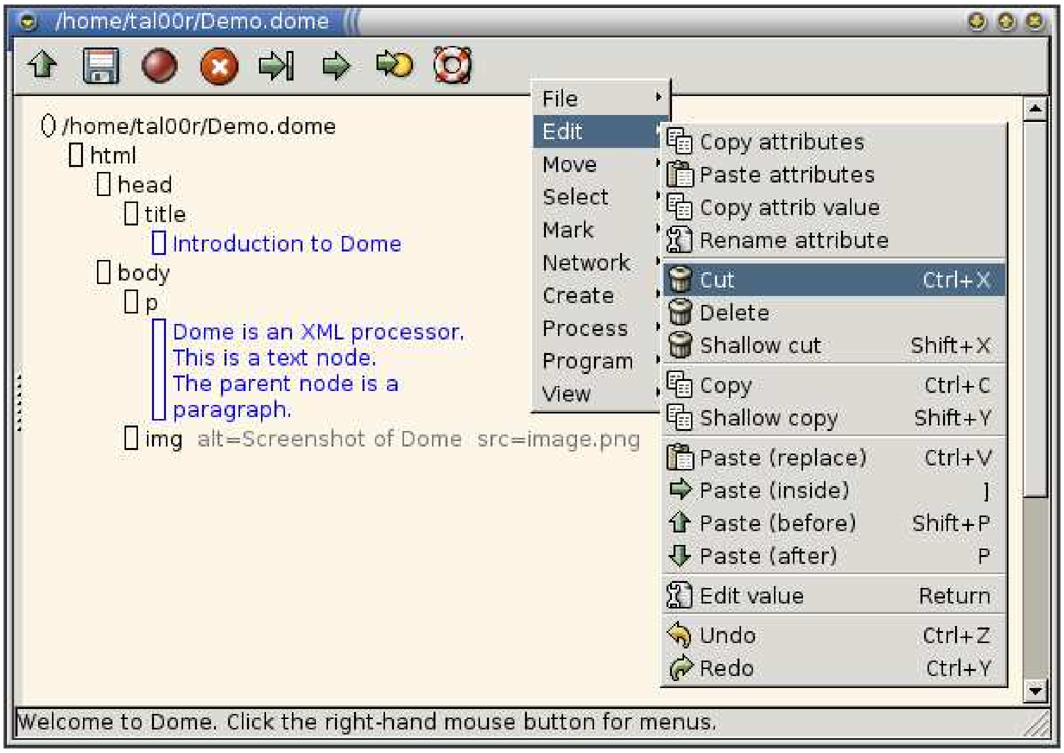


Figure 4.1: A simple XML document being edited in Dome.

4.3 The Dome user interface

The screenshot in figure 4.1 shows the main Dome window displaying an XML document with an `<html>` element as its *document element*. This then contains a `<head>` and a `<body>`. The `<title>` and `<p>` (paragraph) elements contain text, shown in blue. The thin drag bar running down the left edge of the window can be used to pull out the programming controls when required. The interface is designed to look simple and non-threatening.

The editing functions are accessed through the popup menu. Keyboard shortcuts can be bound to menu entries by pressing the desired shortcut while a menu item is highlighted. The toolbar provides common operations (such as Save and Help) and controls for running, pausing and stepping program execution.

Dome provides a wide range of operations to allow the user to specify their intent clearly. This is important in ensuring that recorded operations are replayable. For the same reason, the operations are designed to behave in a consistent way with differing input data. For example, deleting a node always selects its parent node afterwards, rather than its previous sibling (as might be expected). This ensures that the selection will be in the same place afterwards even if the operation is replayed when there is no previous sibling.

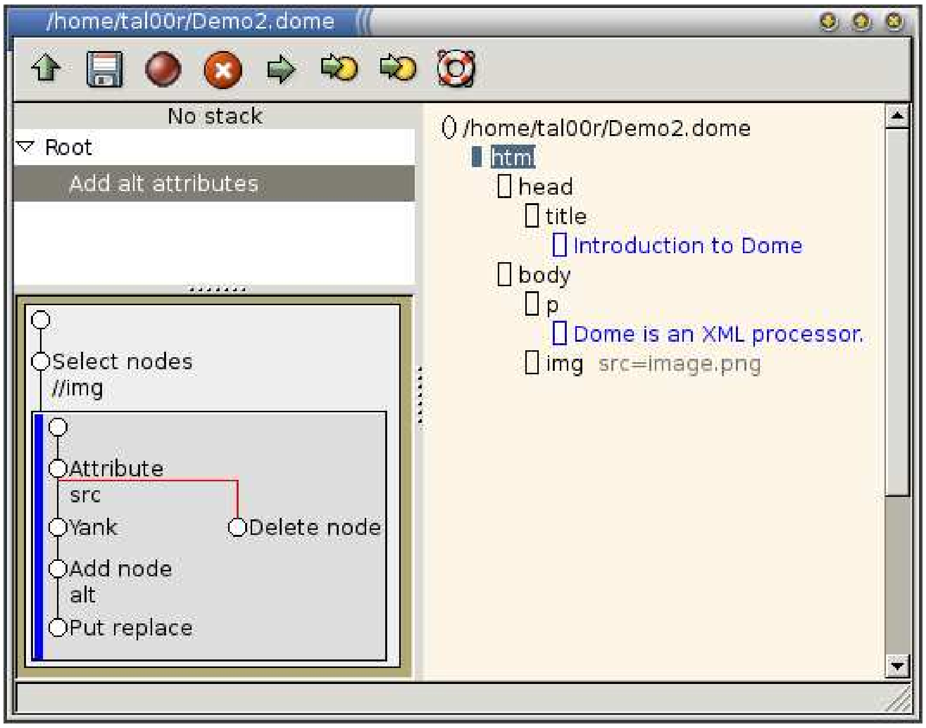


Figure 4.2: Dome's programming interface.

Dome is intended to provide an equivalent to the spreadsheet for trees, and is designed for a similar audience; people who are not professional programmers, but who are competent computer users, around the level of a holder of the *ECDL* (*European Computer Driving Licence*).

4.3.1 Programming with Dome

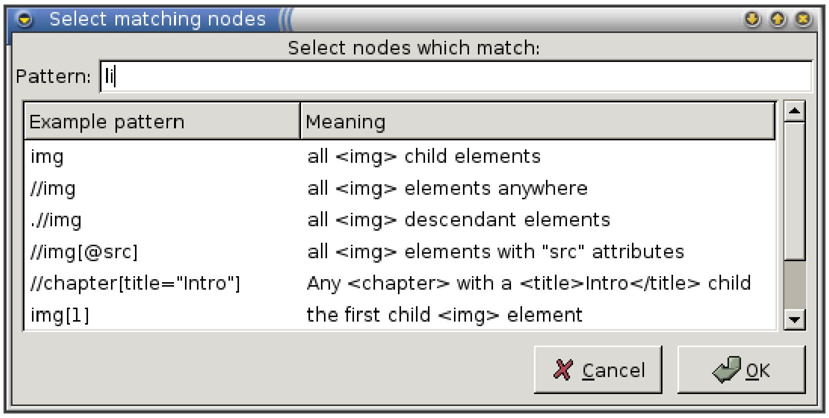
Figure 4.2 shows a Dome session where the user has dragged out the programming sidebar. The upper panel is the list of recorded programs. They are organised in a tree structure, with the default 'Root' program always available. This saves having to explicitly create a new program for simple editing sessions where the solution does not require multiple programs.

The contents of the program which is selected in the list panel are shown in the lower panel. The program shown in the screenshot ('Add alt attributes') sets the '`alt`' attribute of every `<img>` to its '`src`', deleting any image without a source attribute.

Each program is a control flow diagram, with control normally passing downwards along the dark lines. The red branch lines are used when execution of an operation fails for some reason. A yellow dot moves through the diagram, indicating the current execution point. If an error occurs, the position of the yellow dot shows the exact point of failure. A black dot on a line indicates a breakpoint, where execution stops to allow debugging, and the user may also single-step through the program to locate problems.

A Dome program is created by recording normal editing operations. The user can use the program display area to manage the correction of mistakes in recordings and to record alternative sequences to cope with different types of input. While in record mode, a large red square on the diagram indicates the point where any operations performed will be added.

Note that Dome does not have any sophisticated learning algorithms. Instead, a rich set of operations allow the user to specify their intent so that the recordings are reusable. For example, if a user wishes to select all `<li>` child elements, then they should use the 'Select matching nodes' feature and enter '`li`' as the pattern, rather than click on each `<li>` element in turn until all are selected.



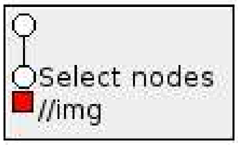
4.3.2 Programs and blocks

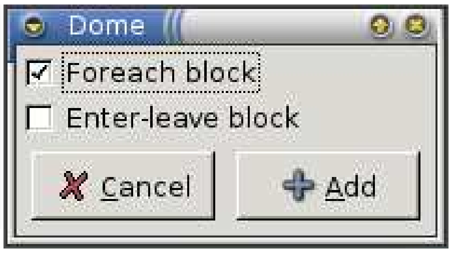
Programs are organised into nested blocks in the familiar *structured programming* style (each block can only be entered at the top, and always exits at the bottom). Blocks are used for looping (the entire block is executed a number of times) and for handling exceptions (an error anywhere within the block will cause execution to continue from a single point at the bottom of the block). They can also be used simply to group sections of code visually, or to make cutting and pasting easier.

To create the above example program, which adds a default '`alt`' attribute to all `<img>` elements in the document from their '`src`' attributes, deleting any which don't have one, the user will start with a sample XHTML document loaded and an empty program (such as 'Root') selected in the program list.

They begin by clicking on the Record button in the toolbar. The red recording square appears in the program display panel to indicate that recorded operations will appear there.

The user now tells Dome to select all the `<img>` elements, by entering the pattern (XPath) `//img`. All such elements are selected. Since the user wishes to perform the following actions on each one they now create a new 'for-each' block:





The new block is created with the red recording point at the start. A blue bar down the side of the block indicates that this is a for-each (loop) block. At this point, just the first of the `<xhtml:img>` elements is selected; we are executing the first iteration of the loop. The user now performs whatever operations are desired on the selected element. In our case, this means clicking on the '`src`' attribute, choosing Copy from the menu, then adding a new attribute called '`alt`' and using Paste to set it to the copied value.

The first iteration of the loop is now complete. The user clicks on the Play toolbar button to continue execution, and Dome processes all the remaining elements in the same way. If the loop is executed for an `<img>` with no '`src`' attribute, execution is paused and recording starts at the point of error (on the red horizontal branch from the select-attribute operation). The user then performs whatever actions are required to cope with that case, such as deleting the element.

Blocks may also be marked as 'enter–leave' blocks. Such blocks are shown with yellow bars at the top and bottom. When the block starts executing, Dome restricts processing to the subtree rooted at the selected node. Thus, if the user had chosen to make the example block above an enter–leave block, it would appear as if the document consisted of only a single `<xhtml:img>` element while the block was executing. This makes the document easier to edit, and makes programs easier to think about (since nothing outside of the selected subtree can be modified from inside the block). When execution of the block is finished, all changes made become a single operation from the point of view of the undo buffer. This automatic grouping of undo operations makes undo easier to use.

A block can be marked as being both an enter–leave and a for-each block, in which case each selected node is 'entered' in turn. For more advanced users, blocks may also be set to save the 'mark' (a secondary cursor) on entry and to restore it on exit. This is used by the XSLT to Dome converter described in chapter 6.

Because looping over finite sets of elements is the standard way of implementing a loop in Dome, infinite-loop bugs are very rare. The need for loops is further reduced due to the fact that most Dome operations apply to all the selected nodes anyway, without the need to loop explicitly.

4.3.3 Comparison with functional languages

XSLT (see section 2.3.3) is a *functional language*[63]. That is, the output is specified as a function of the input, rather than by giving a series of operations to perform to transform the input into the output (as in a traditional imperative language such as Dome). It is worth pausing to consider why Dome takes a different approach to the standard XML transformation language. Indeed, there are functional graphical XML transformation languages already, such as IBM's *XSLerator* (described in section 4.6.1).

While the power of functional and imperative languages is the same, they take very different approaches. Functional languages make it easier to reason about the behaviour of complex systems; for example, a rule that defines what the output document should contain in its `<head>` element cannot affect any other part of the output, no matter how complex its definition.

Since Dome uses a direct manipulation user interface style, familiar to users from other editing software, users are already thinking in an imperative style: programs are created by performing actions. Dome must therefore represent the recorded programs in the same style to make them easily understandable to users. If the operations were converted by Dome into a functional representation, it would force the user to shift mentally between the two styles whenever they switched between creating programs and editing them.

In traditional programming, imperative languages are vastly more popular than purely functional ones, although they have adopted many ideas from them. Often, using an imperative structure to combine a series of functional transformations can bring the best of both worlds, and most languages, including Dome, take this approach to some degree.

4.4 Examples of use

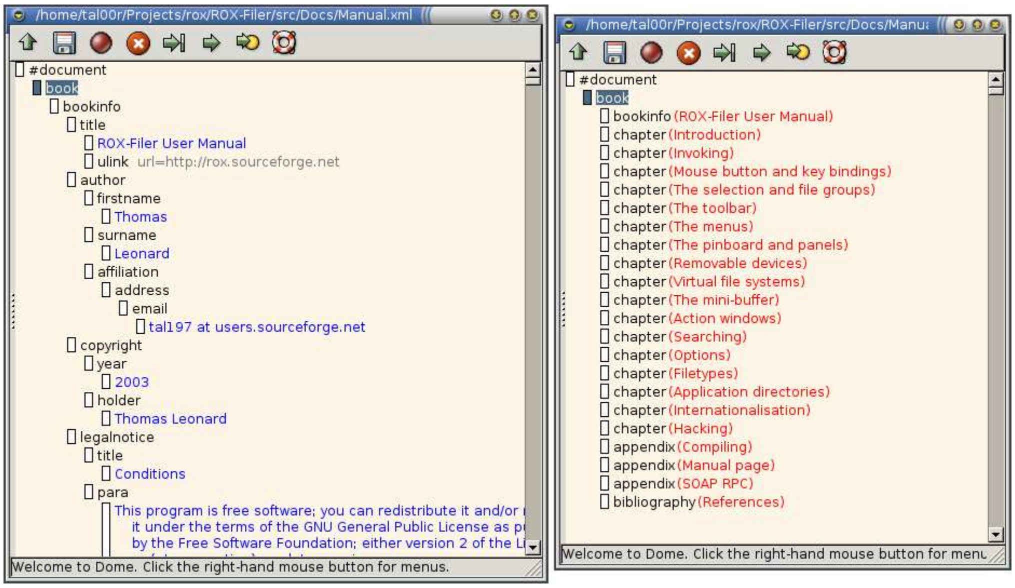
Of the use cases listed above, **Creating web pages**, **Web harvesting** and **Using web services** will be dealt with in chapter 5. The use of Dome to handle the rest (**Viewing and manually editing XML**, **Format conversion**, **Reformatting** and **Querying**) is examined in the following sections.

4.4.1 Viewing and manually editing XML

Dome provides many editing operations, as shown in figure 4.2. Keyboard shortcuts are provided for navigation of the tree, with operations such as ‘next-sibling’, ‘first-child’, ‘parent’ and so on. Because these operations make use of the tree structure, they are able to be recorded and replayed easily. For example, pressing the Down arrow key on a node with no following sibling will refuse to move the cursor, rather than moving to the node that happens to be displayed next. This means that if the move is recorded when there is a following node, and replayed when there isn’t, then the error will be reported.

When a node is selected using the mouse, Dome automatically generates a relative XPath expression to move from the previously selected node to the new one. It is this XPath expression which is recorded when creating programs, allowing the operation to be reliably reused later.

Dome also provides a way to collapse any subtree, similar to ‘folding’ in many text editors. This causes the entire subtree to be shown as a single node. A user editing a long DocBook manual, for example, can get a table-of-contents style display by first selecting all the chapters (using ‘Select Child Nodes’ on the document element) and then using ‘Hide with label’, giving ‘`title`’ as the label. This displays each `<chapter>` as a single element, along with a label taken from each chapter’s `<title>` child element:



Of course, this sequence can be recorded as a program, allowing the user to quickly return to the contents view at any time later with a single click on the program’s name. The user can also double-click on any collapsed node to expand it, and can double click again to return to the collapsed form. The label is actually an XPath expression, and can be used for other purposes too. For example, the XPath '`count(.//*)`' shows the number of elements in each collapsed subtree, rather than the title.

4.4.2 Reformatting

In this section we show how Dome may be used to change the structure of a document, by turning a description list into a table. This is a significant change, since a description list is an alternating sequence of terms and descriptions, whereas a table has each term and description enclosed within a row element. In this example, both the source and the result documents are HTML, but the same process applies to converting between different formats.

To perform the example transformation, we need to move each `<dt>`, `<dd>` pair (*description term* and *description data*) into a `<tr>` (*table row*), putting the term in a `<th>` (table heading) and the description in a `<td>` (*table data*). The source tree will be in this form:

```
<dl>
  <dt>XML</dt>  <dd>The Extensible Markup Language</dd>
  <dt>XSLT</dt> <dd>XSL Transformations</dd>
  <dt>RDF</dt>  <dd>The Resource Description Framework</dd>
</dl>
```

The user begins by changing the root element from `<dl>` to `<table>`. Then, all the `<dt>` elements are selected by entering '`dt`' in the 'Select matching nodes' box, and a loop block is created to process each one, at which point only the first `<dt>` will be selected. A new `<tr>` row element is created above it, and the `<dd>` and `<dt>` elements are dragged inside it. These two elements are then renamed to `<td>` and `<th>`.

The first iteration of the loop is complete. The user now clicks on the Play toolbar button to finish execution, causing the remaining items to be converted. The final output looks like this:

```
<table>
  <tr>
    <th>XML</th> <td>The Extensible Markup Language</td>
  </tr>
  <tr>
    <th>XSLT</th> <td>XSL Transformations</td>
  </tr>
  <tr>
    <th>RDF</th> <td>The Resource Description Framework</td>
  </tr>
</table>
```

Because Dome automatically created XPath expressions to find the destination for the drag-and-drop operations (`preceding-sibling::tr[1]`), they were repeatable for the other loop iterations.

Reformatting in the reverse direction would have been slightly simpler: the `<tr>` elements would be selected and removed (a *shallow delete*), and the other elements renamed. A loop block isn't required here, since renaming and deleting can work directly on multiple elements.

As well as being used for restructuring a document, as shown above, very similar techniques can be used to convert between different file formats, such as turning an RSS news feed into a web page. Dome is also able to load a plain text file as a single XML text node. This can then be broken into lines using the 'Split at...' function with '`\n`' as the separator. This replaces the text node with a `<dome:list>` element containing a subelement with each line of text. The document can then be processed as XML, or each line can be further split into fields if required.

Since Dome is written in Python, more advanced users can process strings using all the features of the Python language. The split operation above could have been performed with the expression '`x.split('\n')`', but common operations such as splitting and search-and-replace have convenience functions for non-expert users.

4.4.3 Querying an XML file

A treesheet can be used as a replacement for *XQuery* (see section 2.3.4). A typical task would be to search an inventory list and locate all out-of-date stock, perhaps returning the result as an order request for new stock.

Since Dome uses XPath for searching, most of the actual complexity involved in making queries is handled there. Selecting records from a database that match some criteria usually involves the user entering an XPath to select the required records, and then using Copy and Paste to move them to a `<Results>` element.

Because Dome breaks the task down into 'select' and 'report' phases, the order is more natural for users. When writing an XQuery, the code for laying out the results is typically written first, with the selection code nested inside it. Dome also gives users instant feedback on the XPath expressions, by highlighting the results in the document. Finally, since Dome generates XPath expressions automatically when any node is clicked, users can click on a node and use the generated expression as a starting point from which to create their own.

4.5 Design evolution

Dome originally grew from visual languages work (see chapter 7), where a flexible visual data-structure was desired. The tree was chosen, and XML was the natural choice for the default format, since so many documents use it. In this section we present our experiences with earlier versions of Dome, detailing some of the problems we encountered and our solutions to them.

4.5.1 Namespaces

XML namespaces are a continual source of problems in the XML world. Consider the case of an XSLT transformation which should load an XHTML document and modify it slightly to produce another XHTML document.

In order to match an element in the source, such as `<table>`, the XPath must contain a prefix. This is because XPath interprets the expression `//table` as a search for a `<table>` element with no namespace, rather than for one in the default namespace, which has been bound to the XHTML namespace.

Thus, we must search for the XPath `//xhtml::table`, and this requires the 'xhtml' prefix to be bound in the XSLT document (the binding in the source document is not used). This in turn causes XSLT to produce the output document elements with this prefix (rather than using the default namespace). While this is perfectly legal for a namespace aware XML parser, the XML DTD schema language used to validate XHTML pages is not namespace aware, causing it to report the page as invalid.

Indeed, scoped namespaces cause a wide range of problems in any environment where searching is to be used, because the prefix bindings vary over the tree. Thus, all the prefixes used in the XPath must be specified as part of the search pattern, which is very clumsy, and searching for `//table` does not do what the user expects.

The problem of non-prefixed XPath matches is such a common source of confusion for users that it is explicitly identified in the requirements for XSLT 2.0[97], section 2.1, and addressed to some extent by the new `<xpath-default-namespace>` element in the current XSLT draft (2 May 2003). This does not actually remove the confusion for users, but does at least make it possible to work around it once you know about the problem.

Previous versions of Dome modified the XPath parser to support default namespaces, but this was confusing since it was inconsistent with other XPath engines. The displayed XPaths could also be misleading if the same prefix (or the default namespace) was bound to different URIs in different parts of the document.

Dome now handles this situation by removing scoped namespaces altogether. Instead, each document has a single (*prefix* → *URI*) mapping. When a document is imported, any bindings used in the new subtree are added to the global list. If there are conflicts, a new prefix is chosen. Dome has a list of well-known namespaces and their traditional prefixes, but will fall back to choosing a random name if the URI is not known, or the prefix is already taken. Further, all namespaced nodes are required to have a prefix; only nodes not in any namespace are shown unprefixed.

This means that the XPath expressions that the user enters or is shown in a program match exactly with what is shown in the document view. Removing scoped namespaces has been found to solve all of the problems with XML namespaces, and we are not aware of any other XML tools that have taken this approach. This may be because most offer both text and tree views of the document, and so must support scoped namespaces in the text view at least. Dome does not provide a textual view of the XML at all, and so has more control over the presentation of namespaces.

4.5.2 Undo and redo

As was mentioned above, Dome's undo and redo is unusual in that it groups operations within a subtree, so that all operations within an enter–leave block appear as a single edit at the next level up.

The original implementation had an even more fine-grained undo system: all operations were stored on the root of the deepest subtree they affected. Any node could be selected, and undo would undo the most recent change within that subtree.

In practice, this was a disastrous interface, since it prevented the user repeatedly pressing undo to step backwards though the recent changes. It was then changed to a traditional global undo. However, this was inconvenient, since there were often hundreds of small changes to step back through, and it was hard to make sure that the state had been wound back to the situation at the start of a particular block in order to rerun the block. Also, storing all the changes used huge amounts of memory; a program which extracted one name from every page in a website ended up storing the whole of every page in the undo buffer.

The enter–leave solution solved both problems. It naturally grouped operations into suitable units for undoing, it only stored the old `<a>` link for each processed page (not the HTML it expanded to, provided the expansion was performed in an enter–leave block) and it was obvious where execution was at any point (ie, at the start of the block which was just undone).

4.5.3 XPaths in Dome

Searching for and selecting nodes is one of the most important operations in Dome. Dome uses XPath[144] expressions extensively for this, and also for calculating values (see chapter 6). XPath expressions are explained in chapter 2, section 2.3.1.

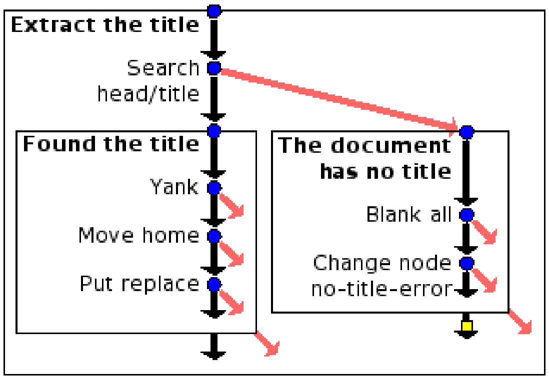
In Dome, relative XPath expressions are created automatically when you click on an element. For example, if the first cell of the first row of an HTML table is currently selected and the user clicks on the first cell of the second row, Dome will record the XPath `../following-sibling::tr[1]/child::td[1]`, which means "Move to the first cell of the next row after this node's parent" [1].

XPaths can also be entered manually, allowing advanced users a very flexible means of locating information, and simpler expressions are often used to select multiple elements (such as `//img` to select every `<img>` element anywhere in the document). An example of a more advanced expression would be `.//*[@uri]`, which selects every node in the subtree below the selected node which has a 'uri' attribute. The role XPath expressions play in searching is similar to that played by regular expressions in a conventional text editor.

4.5.4 Recordings

The program display has gone though many iterations. Originally, a horizontal layout was used, but this took up too much space and required a great deal of scrolling. There were no blocks, but the arcs leading out from a node could be dragged to lead to another node. This visual analog of programming with `GOTO`s suffered from the usual problems of spaghetti code, and blocks were soon added for grouping. Looping was done by 'mapping' a program rather than 'running' it. This required a separate program for every loop block, so the for–each block attribute was added and 'map' deprecated.

The display also used to include stub arrows for unattached arcs, providing a place to click to start recording. This was found to be confusing to new users, and the current implementation only displays a highlight as the pointer is moved over a node. Here is an example of the old interface:



Initially, Dome offered two commands to restrict processing to a subtree — 'Enter' and 'Leave'. Enter allowed the subtree rooted at the currently selected node to

[1]XPath defines a number of shorthand notations; '..' selects the single node on the parent axis and so works much like '..' in a pathname.

be treated as the whole tree until the Leave command was used. However, this made exception handling awkward, since an error during processing in a subtree could cause the 'Leave' to be skipped. Therefore, this functionality became a block attribute — entering such a block triggers the 'Enter' function and leaving the block triggers the 'Leave' in all cases. This also added some useful guarantees about program behaviour, since during processing of an enter–leave block, nothing outside of the selected subtrees can be modified.

4.5.5 Whitespace stripping

Whether the space between elements in an XML file is significant depends on the specific format. In '`<head>  <title>Overview</title>  </head>`', the space between the `<head>` tag and the `<title>` is unimportant. Ideally, Dome should not display this text node at all. However, in '`<a ...>One</a> <a ...>Two</a>`', the space between the two anchors is needed to separate the words. This space does need to be displayed to the user. Ideally, Dome would be able to parse a stylesheet to control the stripping of whitespace on a per-element basis. For now, a text node containing only whitespace will be stripped out when the document is loaded, and extra newlines between elements are inserted when the document is saved (so it is not possible, at present, to run two anchors together).

In a world when XML documents were only ever edited using XML-aware tools, this would not be a problem, since there would never be any reason to insert unnecessary whitespace in the first place. However, many programs insert extra line breaks into generated XML files to make them easier to edit using a text editor. Likewise, users editing XML with a text editor will often create extra newlines to make the display clearer.

4.6 XML editors

There are many XML editors available, but none of them are designed to be programmable. Some provide simple scripting features, offering the ability to write plug-ins in some other language, such as Java, but they do not use the editing operations of the tool itself as the primitives of the language as Dome does. This means that non-programmer users cannot take advantage of these features, and knowing how to use the editor directly gives no insight when programming it.

The tools most similar to Dome are the XSLT generators (such as XSLerator, described below). The typical interface here is to have 'before' and 'after' panes, where the user describes the relationships between them. These tools are very good for certain tasks, but become harder to use when solving more general problems.

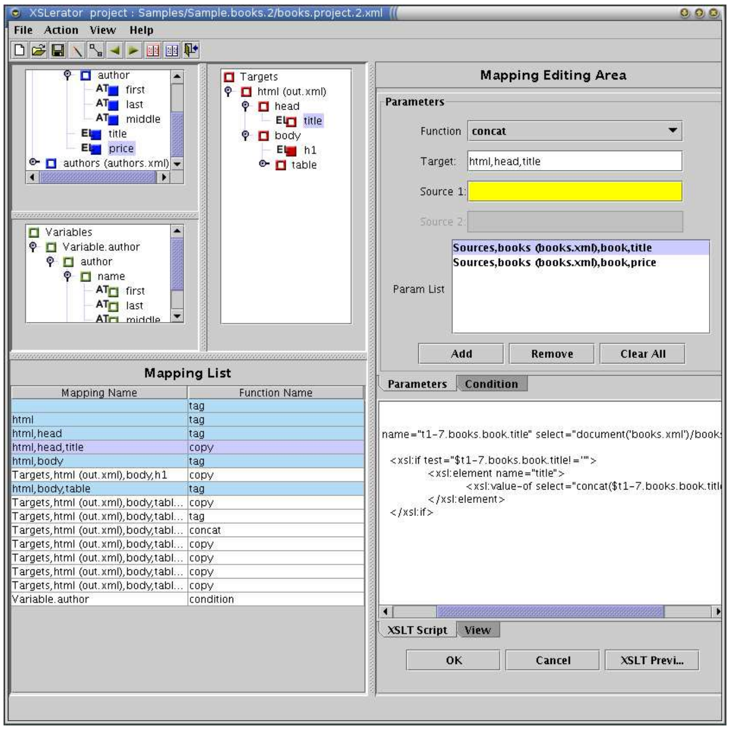


Figure 4.3: The XSLerator interface.

For example, chapter 5 shows Dome being used to collect information from all web pages referenced by a single index page. The source here is the index page, and the result is information from pages it links to. Since the required information is not found anywhere in the source (index) page, there is no direct mapping between the 'before' and 'after' panes. The simple transformation between an individual page on the site and its extracted information can be created easily, but there are no control structures to perform this task for each page listed in the index and to combine the results.

There are also many plain text editors with some form of XML support (such as syntax highlighting, folding or validation). These often provide scripting facilities, sometimes using macro record-and-playback like Dome. However, because the primitives of the language operate on lines and characters, rather than on XML nodes, it is difficult to write reliable programs. Irrelevant differences in white-space, entities, escaping rules and namespaces will all prevent these recordings from being reusable on general XML documents.

4.6.1 XSLerator

IBM's *XSLerator*[64] is a visual tool for building XSLT transformations. First, schemata for the input and output documents are defined (perhaps by loading sample documents, and inferring each schema from them). Then, the user selects each element in the output schema and matches it to an element in the input

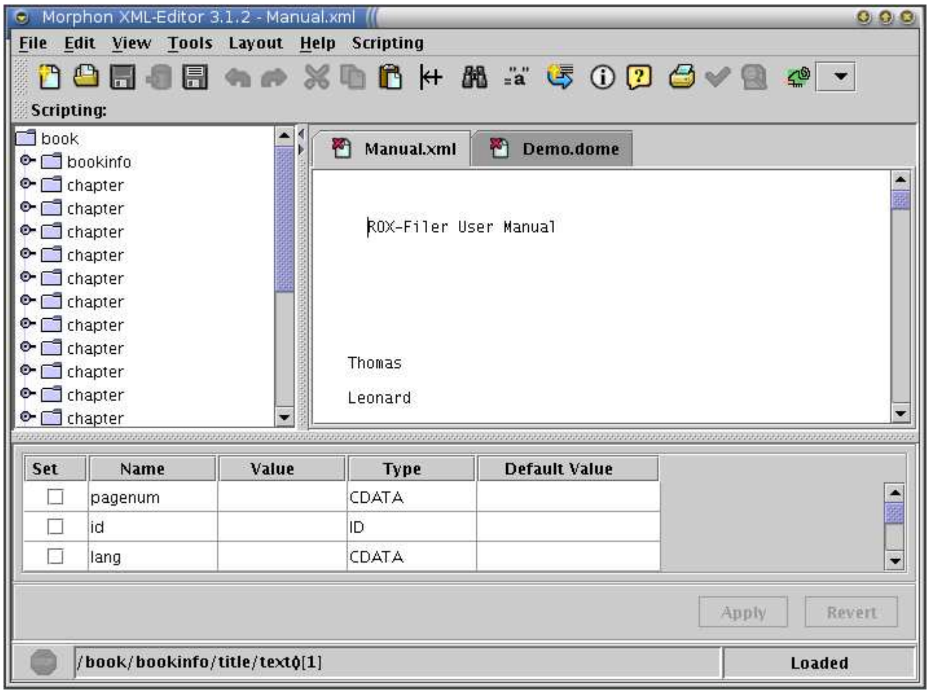


Figure 4.4: The Morphon XML-Editor.

schema. In addition to simply copying elements, the user can choose from a number of other functions (such as concatenating several input nodes to form a single output).

Once the mapping between input and output nodes has been defined, the user can export the mapping as an XSLT transformation. Of course, only a small subset of XSLT's features are available with this system. The interface also has many components (see figure 4.3), with six panes (the source schema, the target schema, the list of mappings, the mapping editor, the list of variables, and the XSLT view).

4.6.2 Morphon

The *Morphon XML-Editor*[95] is a non-programmable XML editor. It displays the element tree and the text nodes in separate panes (see figure 4.4), although the start and end tags can be displayed around the text nodes too if desired. Elements can be folded, as with Dome, but no expression can be given for the summary text, limiting its usefulness (for example, the outline display of a DocBook document is simply a list of `<chapter>` elements, without their titles, as shown above).

The editor can load the DTD for a document, and can present to the user a list of valid attributes or child elements when creating new nodes. Dome does not have this facility, although there is no particular reason why it could not be added.

Another useful feature of Morphon is the built-in CSS editor (see section 3.4.1 for information about stylesheets), which allows very flexible syntax highlighting. This is a considerable improvement over the syntax highlighting offered by traditional text editors, since the structure of the XML makes the rules easy to write and the matching reliable (text editors are plagued by the need to either scan the entire document from the start to determine the correct highlighting at any point, or to use unreliable heuristics).

Morphon also allows the use of XPath for searching (but not for selecting multiple nodes). However, since it allows scoped namespace declarations, it requires the namespaces to be defined explicitly for each search. Dome solves this problem by converting documents to use a global namespace list when they are loaded, as explained in section 4.5.1.

The editor does not provide any scripting ability itself, although it does provide an interface to write plugins in other languages. It cannot, therefore, be considered a tree-sheet, but it is interesting to compare the viewing and editing interface with that of Dome.

4.6.3 XML Spy

XML Spy[2] is a popular XML editing suite for the Windows platform. It is heavily focused on data entry, using the document's schema to prompt the user for the appropriate attributes and child elements as the document is edited. It also has a graphical XML Schema editor, which makes schema definitions much easier to create.

In addition to the ability to collapse any subtree, like Morphon, it offers another interesting method of saving display space: if an element contains a sequence of repeating structures (for example, a `<company>` element which contains a sequence of `<person>` elements, each of which has `<name>`, `<office>` and `<phone>` elements), then that subtree can be switched to a database-like *table view*, with the fields ('name', 'office' and 'phone') along the top, and each row corresponding to one `<person>`.

A separate program in the XML Spy family is the *StyleVision* stylesheet generator. The user constructs the output document, dragging in elements from the schema where values should be dynamically inserted. The editor constructs suitable XPath expressions automatically when the transformation is exported as XSLT.

The program is not able to import XSLT documents, and the interface is focused on creating HTML and PDF output, rather than on performing general purpose transformations. The use of XPaths is generally hidden, although they are used for the test expression for conditional output. The XPaths are modified in the output to insert namespace prefixes, to work around the problem of default namespaces (see section 4.5.1 for the solution used by Dome).

4.7 Conclusions

We began the chapter by identifying the need for a *treesheet*; a tool, analogous to the spreadsheet, which is designed to process tree-structured data such as XML. We have suggested a set of sample use cases for such a tool, and shown that traditional text-based tools are not adequate for these tasks.

We then introduced our implementation of a treesheet, Dome, a visual programmable XML editor. We have described Dome's user interface and some of our experiences with previous versions which led to the current system. We have seen how Dome displays the state of the data in the system visually alongside the programs used to manipulate it. We have looked at the programming constructs that Dome provides, including looping, grouping and restricting processing to a subtree.

We have demonstrated methods of using Dome to meet some of the requirements listed in this chapter, and in the following chapters we will see how well Dome meets the rest of them. In chapter 5 we use Dome to extract information from a large number of web sites, in chapter 6 we compare Dome to the XSLT language, and in chapter 7 we consider Dome as an end-user programming language.

We concluded with a look at some other XML editing tools, and noted some areas for future improvement in Dome. In particular, Dome could provide more support for schema definitions (both to validate the documents it produces and as an aid to editing), and could provide a more flexible range of display styles. However, the other editors we have examined are not suitable for the type of complex programmed tasks for which we designed Dome, being intended instead mainly for manual editing of documents or for the creation of simple stylesheets.

Chapter 5

Dome as a web-harvesting tool

The web is a huge archive of semi-structured information, formatted to allow browsers to display it. Making this available to computer software for bulk processing greatly enhances its usefulness. Typical uses include combining information from multiple sources into a common format, and transforming data into a form which can be placed in a database for querying.

In this chapter we look at the need for a powerful and easy to use harvesting tool, and consider various existing systems. We then show how Dome performs at this task, and present the results of a large-scale web harvesting exercise carried out using Dome.

Most web pages are marked up using *HTML*[135] (the *HyperText Markup Language*), an SGML based format. HTML has a fixed vocabulary, which mixes semantic elements such as `<p>` (paragraph) and `<h1>` (level 1 heading) with display hints such as `<b>` (bold) and `<i>` (italic). Since authors cannot introduce their own elements, information that would normally be marked up with semantic tags such as `<price>` and `<date>` will instead be marked up as bold or italic.

This makes extracting information from web pages a challenging task. The long-term solution to this is for web sites to use semantically-rich XML formats, with stylesheets to transform them for display. However, this is a long way off, not least because many older web browsers would be unable to display the sites, even if web designers started producing them. For many years to come, we will require tools to extract information from these older pages, either to use it directly, or to allow conversion to the newer formats.

XHTML[132] is a direct XML equivalent to HTML. It provides the same limited selection of elements, but enforces XML's stricter syntactic rules. XHTML documents can therefore be reliably read by any XML editor, including Dome. HTML documents are often produced by hand, and are rarely checked for syntactic correctness. In most cases, however, Dome is still able to convert the HTML to XHTML, and then load that.

In this chapter we show how Dome can be used to harvest information from large collections of web pages (either HTML or XHTML) and export the information in a semantically rich form, such as RDF, suitable for storing in databases or for processing with other tools.

Users start by using Dome to process a single sample page from the target site. The operations performed on this source document are recorded, allowing them to be reused on similar web pages. The operations may also be reused at a later date on the same page if it changes, in order to keep the information up-to-date. Facilities are provided to batch process a large collection of pages in one operation, during which the recording can be extended to cope with any irregularities that are discovered.

We describe how Dome may be used to extract knowledge from web sites in such a way that the extraction process may be reliably reused. Our approach takes advantage of the fact that pages within a single site, and in particular those containing data which it is useful to collect, are often generated from a database and have a regular layout from which it is possible to extract information automatically.

As Arnaud Sahuguet and Fabien Azavant note in *Looking at the Web through XML glasses*[115] (1999):

> "Today, more than 80% of information published on the Web is generated by underlying databases (however access is granted through a Web gateway using forms as a query language and HTML as a display vehicle) and this proportion keeps increasing."

It could be argued that if much of the data to be collected is already in a database then it would be simpler to use that directly. In practical terms it is often not possible to get direct access to the database, due to the owner's concerns over security, copyright and the Data Protection Act (the web interface probably only exposes a subset of the available data, or the database may be behind a firewall), or because the database owner may be unwilling to invest time in granting access, even if they have no objections to it in principle.

5.1 HTML to XML extraction tools

There are numerous text-based tools available, such as *Web-OEM*[69] and the tool of Hammer, et al[57], as well as fully fledged programming languages (PERL in particular), but these are not suitable for most users. Even for 'professional'

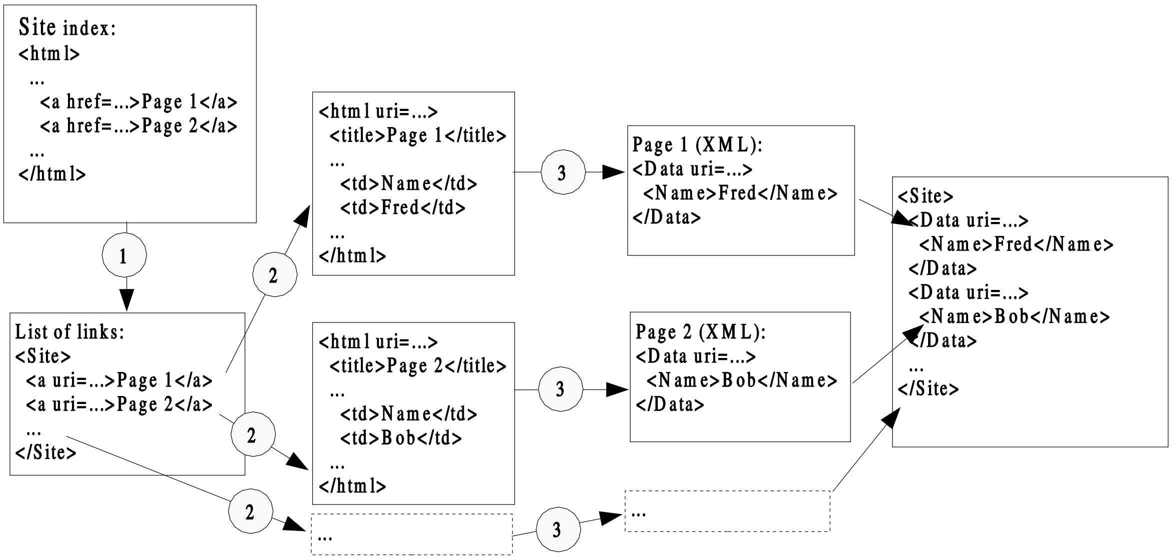


Figure 5.1: Extracting data from a web site's pages

(large-scale) web harvesting, the number of sites to be processed means that requiring specialists to write complex rules to perform the transformation for each site is often infeasible.

Also, the textual tools do not provide a step-by-step view of their execution so that when they fail (perhaps because the web-site has changed slightly) the transformations become difficult to maintain.

A number of other graphical tools exist, besides Dome. *XWrap*[86] and *W4F*[117] are both intended to be used as an HTML to XML preprocessor step in a larger program. *Internet Scrapbook*[127] is aimed at non-expert users who wish to create for themselves a personalised, automatically-updating homepage from other pages on the web. These are discussed below.

The exact process of using Dome for web harvesting is described later, but to aid comparisons with other systems, the steps are briefly listed below (see figure 5.1):

1. Load the site index and extract the list of links to the pages to be processed.
2. Create a loop block which fetches each of these links in turn (Dome provides an operation to replace a link to a web page with the page itself).
3. Convert this HTML into the desired output format. The format chosen, and the subset of the page's data extracted, will depend on the data's intended use. It may be a format the user has made up, or an established format such as RDF (see section 3.4.2) using a semantic vocabulary such as Dublin Core[47].

Once one page has been converted, tell Dome to run the loop block to completion.

The final program can then be run in future to update all the information. Dome has already been used in the way we describe[81], and later in this chapter we give more recent and detailed results.

5.1.1 XWrap

The primary goal of *XWrap*[86] is *"The transformation of "difficult" HTML input into "program-friendly" XML output, which can be parsed and understood by sophisticated query services, ..."*.

XWrap provides an interactive programming-by-demonstration environment where the user indicates regions of interest in an HTML page. Once all regions have been identified, the program generates a Java program to perform the extraction automatically. The XWrap paper identifies four key components in the design, shown here in relation to the equivalent feature in Dome:

Syntactical Structure Normalization fetches a URL entered by the user, cleans up broken HTML and parses the document into a tree.
Dome uses *Tidy*[110] to convert to XHTML and fix broken tags, and *4Suite*[50] to parse the XML into a tree.
The authors mention HTML Tidy and that their code can fix most of the errors that it can, but what the advantages were over using Tidy itself are not made clear.

Information Extraction derives the extraction rules in a declarative form. The user specifies a region of interest, and the program shows the parse tree around that point and allows them to select the tags of interest. Special rules are used for each different element type (bullet lists, tables, and so on).
Dome uses the standard W3C DOM[141] interface, allowing the use of XPath to reference parts of the document.

Code Generation produces an XML version of the transformation.
Dome programs are also XML documents, and may be executed by a non-interactive version of Dome for use in nightly batch jobs, for example.

Testing and Packing allows the transformation to be tried on similar pages and produces a test report.
Dome uses the same interface for both creation and testing. The user can drag in a new source document at any time, and errors detected during bulk processing pause execution, allowing the problem to be fixed in place.

The authors conclude that *"Our initial experience tells us that the main bottleneck in the wrapper generation process is the number of iterations needed to achieve* *a significant coverage of the web site.".* Dome allows the user to switch quickly between different source documents, and to bulk process the site looking for errors, helping to reduce this problem.

5.1.2 W4F

W4F[117, 116], the *World-Wide Web Wrapper Factory*, is quite similar to XWrap; it also generates Java code to perform the extraction demonstrated by the user. Again, the authors seem to have implemented their own parser and, again, mention the problems of fixing broken HTML.

As in XWrap, the extraction rules are stored in a custom declarative syntax; *HEL* (*HTML Extraction Language*) in this case. This is 'DOM-centric', but they use their own notation to reference nodes within it. In this notation, nodes may be separated using '.', which denotes the *child axis* (to use the XPath terminology), or with '`->`', meaning a node following this one in document order. The '`->`' is equivalent to the *descendant axis* plus the *following axis* in XPath.

Like Dome, they support programming-type constructs such as conditions and string manipulation (via regular expressions). Conditions are specified after the path, and are less flexible than XPath predicates (disjunctions, for example, are not supported). Unlike XPath, they support a Prolog-like 'cut' operator to stop a search once a match is found (although, since it is possible to specify a functionally equivalent behaviour in XPath, an optimising XPath compiler could insert a cut itself).

An advantage they claim over XPath is the fork operator ('`#`'). This allows an element to be selected by one path, and then multiple fields extracted from it with sub-paths. This corresponds to Dome's 'enter–leave' blocks, where a particular subtree is selected and then processing proceeds, during execution of the block, as if that subtree were the whole document (see section 4.3.2).

The HTML is shown with a normal browser interface, but each element displays its position in the tree in a tooltip, using the syntax described above. This path is then edited by the user to specify the important tags. The authors note the common problem of marker nodes (such as `<h1>`), which indicate that there is interesting content *following* the element, rather than contained within it.

The tool is aimed very much towards programmers who want to convert HTML to XML before reading it into their own programs.

5.1.3 Internet Scrapbook

The *Internet Scrapbook*[127] allows users to create personal homepages from sections of other sites (for example, they may choose to have today's news headlines from one site and a weather report from another).

The interface is designed to be extremely simple; the user selects the region of interest and then copies it to their homepage. The system uses two heuristics to find the region of interest later when the source page changes:

- The system records the line just above the start of the selection, and the first line of the selection. It uses these to find the start of the region later — first trying an exact match, then looking for the best partial match. The text of the line following the selected region is used in a similar way to find the end of the region.
- Patterns of HTML tags are used as a backup measure if the line-matching algorithm fails, or finds multiple possible regions.

They report that informal experiments with 90 web pages showed the line patterns to be sufficient for 62 of the pages, and using the tags as well they had success with 79 of them. The line matching algorithm came unstuck with diary-type pages where new entries are added to the top, since the program always finds the selected entry as it moves down the page. This is because their interface is not expressive enough to tell the difference between 'keep track of this article' and 'keep track of the most recent article'. In Dome, the topmost article would always be used if the user clicked on it, but the user could perform a text search if they wanted to follow a particular article.

5.1.4 Lixto

Lixto[9, 8] allows the user to specify extraction rules by selecting elements directly from the rendered HTML, in a similar fashion to *Internet Scrapbook*. The user can choose which attributes, such as font style or colour, of the selection should be considered important for matching. Lixto then highlights all matches, and the user can choose to add extra conditions to restrict the items matched by the filter if unwanted elements are selected. These restrictions take the form of requirements that a particular element must appear (or must not appear) before, after or inside the matched element.

Lixto creates *Elog* expressions internally, but the user never needs to see these. Elog is a declarative extraction language which can also be executed by a standalone interpreter. The interpreter generates a *pattern instance base* from the input

HTML, which is then converted to the final XML by a separate *XML generator*, using rules specified by the user.

A Lixto program consists of a number of *patterns*. A pattern contains a number of *filters*, and the pattern matches the union of all elements matched by its filters. Each filter is composed of a number of *conditions*, all of which must succeed for an element to match. Patterns are defined recursively; they will only find matches inside their parent pattern's matched elements. The *document* pattern is a predefined pattern which matches the whole document. Each pattern corresponds to an XML element in the output, nested in the same way as the patterns are.

Lixto makes some changes to the document structure when loading a page, such as inserting implied `<p>` elements (web designers tend to use `<p>` to separate paragraphs rather than to contain them, so the first paragraph in a sequence normally lacks a `<p>` tag), and replacing `<b>` with a `<content>` element with a bold attribute.

Lixto originally used its own path syntax, although normal XPath expressions can now be used. The original syntax is similar, but more limited. For example, `*.table.*.tr` in Elog corresponds to the XPath expression `//table//tr`.

Lixto has a number of special built-in types, allowing it to recognise dates, countries and currencies. Extra types can be defined by means of regular expressions. Like Dome, Lixto is able to extract from all pages referenced by an index page.

5.2 Web harvesting with Dome

Reliably locating information within XHTML pages is the most challenging aspect of web harvesting. Whereas most XML formats use an identifying element name for each piece of information (such as `<date>`, `<price>` or `<name>`), information in XHTML documents is only marked up with generic elements, such as `<b>` (bold) or `<i>` (italic).

Since several pieces of information may be marked up with the same tags, or may not be marked up at all, the choice of method to locate each item greatly affects the probability that the extraction can be replayed successfully on other pages. There are four common ways in which a Dome user selects a node in a document:

1. Any node in the document may be clicked with the mouse, causing Dome to create an XPath expression to select that node, relative to the previously selected node.
2. A node may be selected, as above, but with the additional requirement that the text of the node must match too.
3. A plain string, such as 'Name' may be searched for anywhere in the page.

4. An XPath expression may be entered directly.

Although all four methods may be used to select the same node, choosing the correct method helps ensure that the operation is replayable. Experience has shown that the best results are obtained by first searching for some identifying text, such as 'Name', and then clicking on the actual data element. Although searching could find the wrong node, the relative move will almost always cause the error to be detected.

The second method is generally only useful for finding information on very rigidly structured pages, but it is useful to create 'if'-type branches when you have an example page with a particular element, but wish to perform a different action if, for example, the text 'Student' is found instead of 'Professor'.

The last method is most useful for selecting multiple nodes in one operation. Although any XPath can be used, most cases are covered by only three patterns: '`element`' to select all child `<element>` nodes, '`//element`' to select all `<element>` elements anywhere in the document and, less commonly, '`.//element`' to select all `<element>` nodes in the selected subtree.

5.3 Processing one page

In a typical editing session, the user will start with a web page from the site of interest. They will then record a program to extract just the information they require. For example, if the aim is to extract the name, price and order code for various products, the user first creates some empty elements at the top of the document: `<name>`, `<price>` and `<order-code>`. They then scroll down to find the product's name and drag it to the `<name>` element (cut and paste could also be used). This is repeated for the price and the order code. Finally, the original HTML is deleted, leaving a neat XML record to be saved out.

To extract information from a similar page, the user can load the page in and click on the recorded program to run it. While testing the program on other pages, the program may fail (for example, if one product has no order code because it is unavailable at the moment). Playback stops at the operation corresponding to the user clicking on the order code to begin the drag, and the user is prompted to switch to record mode. The user performs some suitable actions for this case, such as inserting the text 'Unavailable' in the `<order-code>` element, as shown in figure 5.2.

Note that no manual editing of the program was required to create the example, except that the operations for the order code were grouped in their own block (by dragging a box around them). Whichever path is taken, execution resumes after

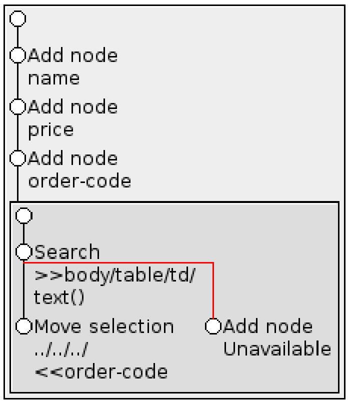


Figure 5.2: A Dome program which extracts an order code, if present, or stores the word ‘Unavailable’ if not.

the block, which is useful if there are more operations following (for example, the `<price>` still needs to be extracted whether there was an order code or not).

5.4 Processing a whole site

Processing a whole web-site requires following hypertext links. A piece of text in an XHTML web page is marked as a link by placing it in an `<a>` (anchor) element. The anchor element has an ‘`href`’ attribute giving the *URI Reference* of the other end of the link (see section 3.3).

Dome provides a function to fetch a web page referenced by another page. It does this by replacing the anchor element node with the contents of the page fetched (in the case of XHTML, this is a subtree whose root element is `<xhtml:html>`).

Since Dome allows any element in the document to be treated, temporarily, as the document root (see section 4.3.2), it is possible to ‘enter’ the new subtree and run the program previously created, which will process the page as described in the previous section.

Therefore, to process an entire site, the following steps are typically used (see figure 5.1):

1. Fetch an index page which references all the subpages to be harvested.
2. Extract all the relevant ‘anchor’ nodes from that document and select them.
3. Create an ‘Enter–Leave–Foreach block’ to process each one (see section 4.3.2). When the block is executed, Dome runs the code inside the block once for each selected anchor element, with that node appearing to be the root node.
4. Inside the block, the referenced page is fetched and the program for processing one page run. Thus, the effect of running the whole block is to replace each reference to a page with the information extracted from the page.

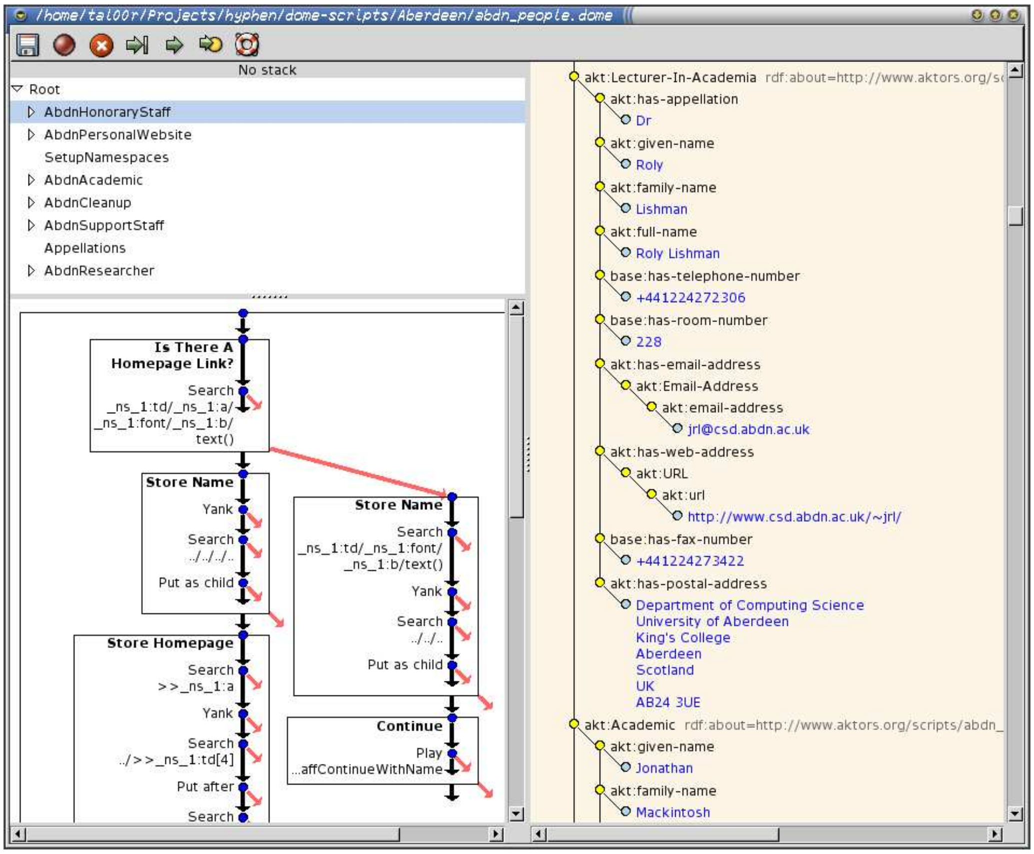


Figure 5.3: A web-site's information extracted and converted to RDF using Dome.

This generates an XML document which is a list of page references and their extracted information. If any of the pages cannot be processed, execution stops and the user may record an alternative branch to cope with the problem, as described above. Execution can then be resumed from that point.

When extracting data from the web, Dome automatically records the URI of each page fetched by setting a '`uri`' attribute on the root of the fetched sub-tree. In addition to being a useful reminder of where the data came from, the attribute is used to resolve relative URIs within the document, allowing further pages to be fetched recursively.

When batch processing a whole site using the non-interactive version of Dome, an untrapped error will halt the extraction and notify the user (eg, by email). It is also possible to record a failure case for a failed HTTP GET or POST operation which replaces the anchor node with an error marker and continues with the rest of the document. A refinement of this is to leave the '`uri`' attributes on the error nodes, then select all the error nodes at the end and try to process them a second time so that intermittent faults have less effect. It is also possible to generate a problem report document from these links, which lists the pages which cannot be fetched.

5.5 Exporting the results

If some output format other than plain XML is required, another Dome program may be used to convert to that format. More usefully, several programs may be employed to export the same data in a variety of formats. Typically, therefore, the process is broken down into two steps: a single program produces minimal XML records containing just the extracted information, then one or more extra programs convert this to other formats.

For example, a list of XML records may be converted to an XHTML table as described in section 4.4.2. Once the other elements, `<html>` and `<body>`, are added to the document, it is ready to be published on the web.

Dome also provides an XHTML-to-HTML converter. This can be used to create web pages suitable for older browsers, once a Dome program has created the XHTML from the extracted information.

For use in knowledge systems, records may be converted to the RDF format (see section 3.4.2), as shown in figure 5.3, perhaps using a semantic vocabulary such as Dublin Core[47]. Note that the screenshot shows an older version of Dome.

5.6 Evaluation

As part of the *AKT project*[120] (*Advanced Knowledge Technologies*), three undergraduates were employed to extract information about computer science researchers from other UK university web sites, and this opportunity was taken to test Dome. In this section we report their experiences learning and using Dome, and the longer term process of keeping the information up-to-date.

They initially created Dome recordings to fetch a departmental index of people from three sites and to extract information from each of the linked pages. The information was converted to RDF (still using Dome). The Dome programs they created were run automatically every night for several months to update the information.

The information extracted included each person's telephone number, group, web-page, email address, given name, family name, full name and appellation (where available). The three initial sites were Aberdeen, Edinburgh and Sheffield.

The undergraduates were employed for one week, which included learning to use the early version of Dome, becoming familiar with the page structure, and gaining an understanding of the RDF required. They all completed the tasks well within the allotted time. Shortly after this, one of them created a program for Imperial College. This took two hours, using a newer version of Dome.

A few months later, they were employed again to extend this to 12 sites, and to include more information (such as publications and projects), creating a total of 50 Dome programs. These programs were run automatically each night, and the information is available via the web at `http://hyphen.info`.

5.6.1 Usability

One of the main complaints about the system was its speed when used interactively. Many parts of Dome were rewritten during and after the experiments, and interactive responsiveness is no longer a problem. The two main factors contributing to this were a move from a pure Python library implementing the XML DOM interface to one written partially in C, and a redesign of the redraw code which makes the redraw speed proportional to the number of elements displayed, rather than to the total number of elements in the document.

The undergraduates also reported that some of Dome's hidden state information caused confusion, such as whether the display showed the whole document or only a subtree. Extra visual indicators were added to provide this information.

5.6.2 Maintainability

In the months since the original recordings were made the sites had their information updated and this was correctly handled by the scripts without any modifications. *Wrapper induction: Efficiency and expressiveness*[76] reports that the mean-time between structural changes in a web-page is about one month, but the pages used in our sample remained stable for much longer than this, perhaps because they were not commercial sites. Both Dome and some of the pre-release versions of certain libraries that it used have undergone a number of incompatible changes since the experiments were performed, and this actually caused more compatibility issues than changes to the sites.

5.6.3 Processing speed

While Dome cannot compete with most XSLT engines when it comes to performance, it is reasonably fast at bulk processing sites, and certainly adequate. The machine used for the nightly updates was a standard Pentium III 600MHz computer.

Since the creators of the scripts were novice Dome users, the programs were not as efficient as they could have been and could be improved in many places. For example, Dome allows all selected elements to be renamed in a single operation, but a novice user may not realise that this is allowed and create a loop block to rename each one in turn, which is considerably slower.

	Time (m:s)	People	RDF triples
Aberdeen	2:21	55	446
Edinburgh	2:07	446	3730
Sheffield	6:48	132	1088
Imperial College	3:47	257	5363

Table 5.1: Extracting data from a web site's pages

Table 5.1 shows the running time of each script, the number of people about whom information is collected, and the total number of RDF triples[1]. The timings were extremely variable (up to 40 minutes some days). The best time for each site is shown, as other values indicate network problems.

To address the scalability issues, the 'enter–leave–foreach' block structure could allow each selected subtree to be automatically processed in parallel, without modifications to the existing scripts. This may help to reduce time spent waiting for data from the network, and allow us to take advantage of multiple processors. However, the speed improvements gained by the faster XML library have made such optimisations unnecessary at present.

Another attempt to improve scalability involved getting Dome to record the MD5 sum of the contents of each page fetched in an attribute. This also allowed the Dome program to report which pages had changed. This could be useful, since even if the output XML was the same, a change to a significant number of pages could indicate the addition of new information to extract. The Last-Modified date from the HTTP header was also used, when given, but was often not provided (especially for dynamically generated pages where the modification date of the most recently changed database item would have to be calculated each time).

Recording this information about the source page meant that the output of the previous run could be used as the new input, and only pages which had changed needed to be reprocessed. This was used for a number of the scripts, but it introduced extra complexity and opportunities for failure. Again, the other speed increases meant that processing time was no longer the major bottleneck, and this feature is no longer being used.

5.7 Web searches and queries

So far in this chapter we have looked at fetching pages to which we have an explicit link. However, much of the information available through the web is accessed

[1] A triple consists of a subject, an object and a relationship, for example "person-x has-email-address address-y".

through HTML forms or web services (see section 3.2.3), where the reader specifies some query terms. In this section, we show how Dome can access these pages.

Accessing information through forms is done by requesting the search results page, but passing the query terms along with the URI of the page. HTTP provides two ways to send this information. The first is to encode the fields in the URI itself (an *HTTP GET* request):

```
http://www.google.com/search?&safe=off&q=life
```

The above URI identifies the result page for a Google search for 'life', with 'safe' mode set to off. As far as Dome is concerned, this is a normal URI; all special processing is done by Google's servers. The URI can be created using Dome's string manipulation features. Given a text node with a search term (such as 'life'), the user inserts before it another text node with the value `http://www.google.com/search?&safe=off&`. The two nodes are then joined and the result is used as the URI for a fetch. Users can easily see what the final URI should look like by performing a search manually with Google and looking at the URI displayed in the location bar.

The other method of accessing web forms is an *HTTP POST* request. Here, extra $(name, value)$ pairs are submitted along with the URI, but are not a part of it. These fields do not appear in the URI that the user sees. When Dome's 'HTTP POST' feature is used, all non-namespaced attributes of the anchor node are sent as extra POST fields. Although the fields are not shown in the URI, users can still discover them by using the 'View Page Info' feature provided by many browsers, which shows a list of fields and values when invoked on the results page.

In both of the above cases, the user interface is specified using HTML forms, and the values submitted are plain text strings. In August 2003, the Web Consortium's *XForms*[134] specification became an official recommendation. An XForms submission sends the query as an XML document, rather than as a set of named strings, and so fits more naturally into Dome's method of working. A template for this document is typically given in the `<head>` part of the HTML query page, while the layout (which is irrelevant to Dome users) is given in the body. This provides a simple template which a Dome user can fill in and submit. However, support for XForms has not yet been implemented in Dome since XForms is too new to have seen any real-world adoption.

For more advanced users, Dome can also be used to access SOAP-based web services. The user must manually create a SOAP message and then use Dome's 'Send SOAP Message' feature, which replaces the SOAP message subtree with the SOAP result (in the same way that fetching a web page replaces the anchor element with the fetched page). Typically, accessing a site's features using this method is more work (especially as many sites require registration first, and require an identifying key to be sent with each request), but the result document will be easier to parse, since it is designed for machine reading.

5.8 Conclusions

In this chapter we have shown how Dome may be used to extract information from web pages to create machine-readable RDF documents. We have seen how to process many pages automatically and have looked at ways of making the extraction process robust to changes in document structure. The system has been used to regenerate tens of thousands of RDF-triples from HTML sources each night for a period of several months.

The extraction process is simple enough that users can quickly learn to use it, and powerful enough to solve complicated problems, such as merging data from several pages, getting information through online HTML forms, and coping with differing page structures.

The initial experiments revealed some speed problems with both interactive and batch use, but these have been addressed through the use of much faster library code, and other optimisations to Dome itself. There are still many possibilities for further improvement, such as taking advantage of enter–leave blocks to parallelise programs automatically, but given current performance such techniques are not presently required. None of the test subjects had problems understanding Dome's operation, or in completing their tasks.

In its current state, we consider Dome to be a useful tool for anyone wishing to process web pages in a structured and repeatable way, and we hope to be able to repeat the experiments at some point using the new version of Dome.

Chapter 6

Comparison with XSLT

Dome, introduced in chapter 4, is a transformation language for XML documents. In this chapter we compare Dome with XSLT, the Web Consortium's official transformation language for XML. XSLT is a textual language in which the user defines rules to create an output document from an input document.

We start by giving a brief introduction to XSLT, and its predecessor DSSSL. We then compare idiomatic Dome and XSLT solutions to the sample processing task of generating XHTML from an XML source document. Finally, we demonstrate how Dome is able to import XSLT programs, and convert them automatically into Dome programs.

6.1 XSLT — XSL Transformations

XSLT[36], briefly introduced in section 2.3.3, is a language for transforming XML documents. It was originally part of *XSL*[1] (the *Extensible Stylesheet Language*), which adds formatting information to documents. For example, a typical transformation might replace each `<author>` element in the source document with a `<fo:block font-weight="bold">` element in the transform's output. XSL was later split into *XSL Formatting Objects*, providing a vocabulary for expressing document formatting, and *XSL Transformations* (XSLT), a general purpose transformation language.

When used to convert a document to the *XSL Formatting Objects* model, the output from the XSLT transform is then processed by a *formatter* to produce the final output in each desired format, such as PDF (*Portable Document Format*) or HTML. Separating the transformation stage (which creates abstract formatting information) from the generation of the binary file allows a clean separation between the desired style (selected by choosing from a number of XSLT transformations) and the desired output file format.

This two-stage approach is based on *DSSSL*[67] (see section 6.1.2), and the formatting is also heavily influenced by *CSS*[15] (*Cascading Style Sheets*, described in section 3.4.1).

> "XSLT is also designed to be used independently of XSL. However, XSLT is not intended as a completely general-purpose XML transformation language. Rather it is designed primarily for the kinds of transformations that are needed when XSLT is used as part of XSL."
>
> — XSLT Specification[36]

Despite the modesty of the authors, XSLT is actually an extremely flexible language for specifying complex transformations.

6.1.1 XSLT syntax overview

An XSLT transformation is a set of *templates*. Each template matches a particular set of nodes in the source document, and contains some content to create in the output document. When a stylesheet is applied to a source document, the XSLT processor finds the template which matches the document node[1] and *instantiates* it to create the result document.

For example, if we apply this stylesheet ...

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result>The old text was ‘<xsl:value-of select=’/document’/>’</result>
  </xsl:template>
</xsl:stylesheet>
```

... to this source document ...

```
<?xml version="1.0"?>
<document>Some text.</document>
```

... then we get this result document:

```
<?xml version="1.0"?>
<result>The old text was ‘Some text.’</result>
```

Instantiating a template can cause other nodes in the source document to be processed (the most common case being to process all the child nodes of the source node). The default rule for element nodes simply causes all the source's child nodes to be processed recursively.

[1] The node representing the document itself, which contains the *document element node*.

A template's content is not restricted to a single element; it can also output only text, or a sequence of element nodes. Therefore, the output of an XSLT transform is a *document fragment*, not necessarily a document directly serialisable as well-formed XML.

The '`match`' attribute specifies which nodes the template can be used on. It is an XPath expression, so nodes may be selected based on many criteria, such as their attributes, content, or position in the tree.

As shown above, the content of a template is a mixture of *literal* nodes (such as `<result>` and `The old text...`) and special XSL elements which cause further processing. Examples of these are the `<xsl:value-of>` element (which evaluates an XPath expression and inserts the result into the output tree) and the `<xsl:apply-templates>` element (which selects other source nodes and processes them in a recursive fashion).

6.1.2 History

DSSSL[67], the *Document Style Semantics and Specification Language*, is an older system for transforming and formatting (SGML) documents, and has strongly influenced XSL. Since XML is a subset of SGML, DSSSL can be used to format XML documents. It consists of four parts:

A transformation language The source SGML document is first transformed using rules specified using the transformation language to give a new SGML document, which is used by the formatter. [2]

A style language The formatter takes the new SGML document, output by the transformation step, and applies the rules specified using the style language to generate the *flow object tree*. This tree is then converted to the final output format (for example, PDF or HTML).

A query language The Standard Document Query Language is used to identify parts of the source document, playing a similar role to XPath in XSLT.

An expression language DSSSL's syntax is based on the *Scheme*[65] language (specifically, the functional, side-effect free subset of the language), and it supports similar features (functions, conditionals, and so on). Apart from the syntax, it is quite similar to XSLT; a DSSSL stylesheet defines a number of rules for matching elements in the source document, and the *SOSOFO* (*Specification Of a Sequence Of Flow Objects*) to create for each rule.

DSSSL also supports different *modes* (named collections of rules), like XSLT, so that the same element can be processed in different ways depending on context.

[2] In fact, as with XSLT, there may be multiple source documents and multiple result documents.

For example, a `<title>` source element will be formatted one way when it appears in the table of contents, but differently when the same element is used to start a chapter.

A typical DSSSL rule looks like this:

```
(element p (make paragraph font-size: 10pt (process-children) ))
```

This causes each `<p>` element in the source document to be output as a paragraph with a font size of 10pt. It roughly corresponds to the XSLT template:

```
<xsl:template match="p">
  <fo:block font-size="10pt">
    <xsl:apply-templates/>
  </fo:block>
</xsl:templates>
```

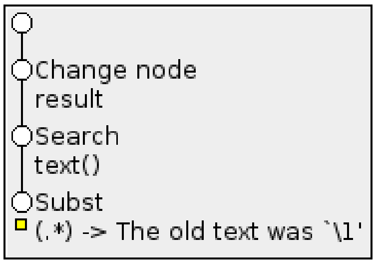
There are a number of DSSSL processors available, including the free programs *Jade*[72] and *OpenJade*[6].

6.2 Dome and XSLT

Starting with the example given at the start of this chapter, we can compare these two implementations of a transform which renames the document root and changes the text inside it:

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <result>The old text was ‘<xsl:value-of select=’/document’/>’</result>
  </xsl:template>
</xsl:stylesheet>
```

The Dome recording on the right was created by editing the root node directly, then clicking on the text node and performing a regular-expression search and replace.



6.2.1 *Processing a FAQ for a website*

The task for this comparison is to generate an XHTML FAQ with a table of contents. First, the existing HTML FAQ, which did not have a table of contents, was converted to more structured XML (using Dome).

This XML document will be edited to update the FAQ. Afterwards, it will be processed to create the XHTML document. We start by just generating the

XHTML without the table of contents. This sample illustrates the format of the XML source:

```
<faq>
  <section><title>General questions</title>
    <entry><question>Why doesn't it work?</question>
      You need the <xhtml:strong>new</xhtml:strong> version.
    </entry>
  </section>
</faq>
```

Sample solution using XSLT

The first part of the transformation tells the XSLT processor that we wish to write out XML, and that the document element (`<faq>`) should be replaced by `<xhtml:html>`:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version='1.0' xmlns:xhtml="http://www.w3.org/1999/xhtml">

  <xsl:output method="xml" encoding="utf-8" indent="yes"/>

  <xsl:template match='faq'>
    <xhtml:html>
      <xsl:apply-templates/>
    </xhtml:html>
  </xsl:template>
```

The next rule creates an `<xhtml:div>` element for each `<section>` in the source. The contents of the `<title>` element is placed into an `<xhtml:h3>` to create the section heading, and then each child `<entry>` node is processed:

```
  <xsl:template match='section'>
    <xhtml:div>
      <xhtml:h3><xsl:value-of select='title'/></xhtml:h3>
      <xsl:apply-templates select='entry'/>
    </xhtml:div>
  </xsl:template>
```

Each entry in the FAQ has a `<question>`, which is placed in an `<xhtml:h4>`. Everything else in the element is copied directly to the output (it is already marked up in XHTML):

```
  <xsl:template match='entry'>
    <xhtml:h4><xsl:value-of select='question'/></xhtml:h4>
    <xsl:apply-templates mode='all'/>
  </xsl:template>
```

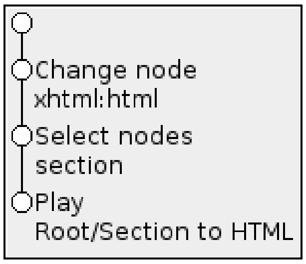
Because the XSLT built-in templates default to removing all elements and attributes, we need an extra rule to copy them across (this part is taken directly from an example in the XSLT specification):

```
  <xsl:template match='@*|node()' mode='all'>
    <xsl:copy>
      <xsl:apply-templates select='@*|node()' mode='all'/>
    </xsl:copy>
  </xsl:template>
```

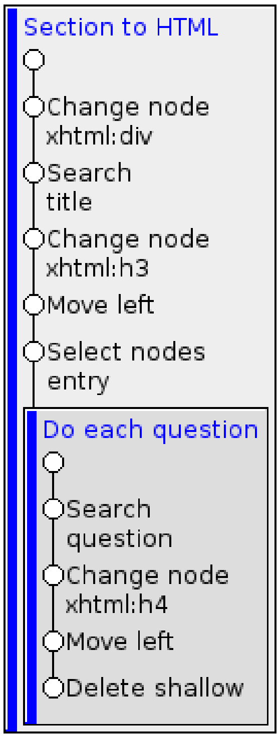
The final rule prevents the `<question>` from being copied directly to the output:

```
  <xsl:template match='question' mode='all'/>
</xsl:stylesheet>
```

Sample solution using Dome



On the left is the main program. It changes the root node from `<faq>` to `<xhtml:html>`, selects all of the `<section>` elements, and runs the 'Section to HTML' program on all of them.



The bulk of the work is done in this second program, shown on the right. After creating a new empty program, the user made it a 'foreach' block and ran it while still recording the main program. Dome started recording in the new program (since it was empty), with the first of the `<section>` nodes selected (since it was the first iteration of the loop). The user changed the `<section>` into an `<xhtml:div>`, clicked on the `<title>` and changed it to an `<xhtml:h3>`, then selected all the `<entry>` elements and created a new loop block to process each one.

At this point, the first `<entry>` was selected. The user changed the `<question>` to an `<xhtml:h4>` and removed the `<entry>`. Although only one entry had been processed, the program was finished. Clicking on the Play button ran the whole job to completion.

6.2.2 Adding the table of contents

We need to generate a unique ID for each question so we can create a link to it. We will use the XPath expression `count(preceding::question)` to number each question (by using the length of the list of `<question>` elements that appear earlier in the document as the ID) [3].

Table of contents in XSLT

Here is the modified code which writes out the question heading to include the ID for the `<question>` element:

```
<xsl:template match='entry'>
  <xhtml:h4 id='{count(preceding::question)}'>
    <xsl:value-of select='question'/>
  </xhtml:h4>
  <xsl:apply-templates mode='all'/>
</xsl:template>
```

In the XSLT template which processes the document element of the source, we create a heading and then process each of the sections twice; once in 'toc' mode (generating table-of-contents entries) and once in the default mode (generating question and answer pairs):

```
<xsl:template match='faq'>
  <xhtml:html>
    <xhtml:h2>Contents</xhtml:h2>
    <xsl:apply-templates mode='toc'/>
    <xsl:apply-templates/>
  </xhtml:html>
</xsl:template>
```

Each section, when processed in 'toc' mode, will produce a section heading and then process each of the `<question>` elements, also in 'toc' mode, placing the results inside a numbered list:

```
<xsl:template match='section' mode='toc'>
  <xhtml:h3><xsl:value-of select='title'/></xhtml:h3>
  <xhtml:ol>
    <xsl:apply-templates select='entry/question' mode='toc'/>
  </xhtml:ol>
</xsl:template>
```

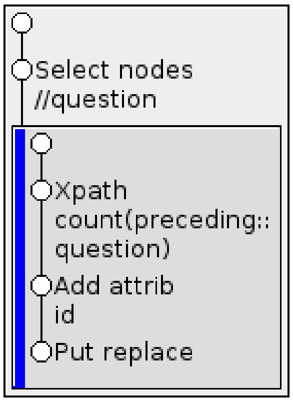
[3] XSLT supports the 'generate-id()' function, an extension to XPath's standard functions which creates a unique ID for each source node. The comparison to Dome is easier if we stick to standard XPath, however.

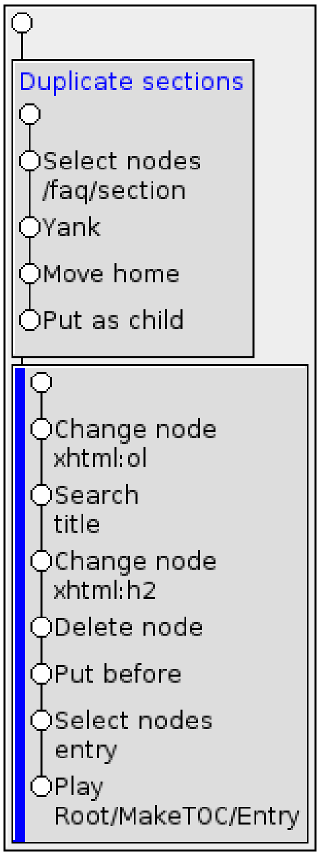
Finally, the template for processing each individual question places its text inside an anchor inside a list item. The target for the <a> is the ID of the <question>, as before:

```
<xsl:template match='question' mode='toc'>
 <xhtml:li>
  <xhtml:a href='#{count(preceding::question)}'>
    <xsl:value-of select='text()'/>
  </xhtml:a>
 </xhtml:li>
</xsl:template>
```

Table of contents in Dome

In Dome, we use a different technique. First, we select all the question elements and loop over them, adding numbered ID attributes, as shown in the program on the right. The ID number is calculated as in the XSLT and the result, which is placed on the clipboard, is then pasted into the new attribute.

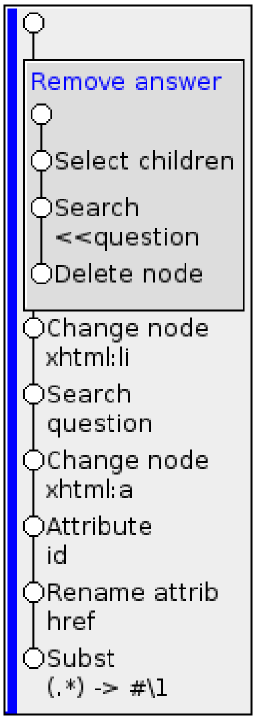




Whereas in XSLT we processed the source document twice, in Dome we make a copy of the source sections and trim one copy down to just the questions (as shown in the program to the left).

In each section, we turn the <section> into an <xhtml:ol> (ordered list), turn the <title> element into a heading, and place it just before the list. We select all the elements inside the section and use a third program (below) to turn each <entry> into a link to the question and answer pair.

For each entry, we select everything (question and answer), then unselect the question (by Ctrl-clicking on it) and delete. The <entry> becomes an <xhtml:li> and the <question> an <xhtml:a>. The 'id' attribute is renamed to 'href' and a leading # inserted.



6.2.3 Comparison

It is clear that there are many similarities between the Dome and XSLT approaches. Both rely heavily on XPath expressions, both for selecting nodes and for calculating values.

Because Dome can be used to modify the existing document in-place, it is often easier to use when making a number of small changes. Also, it makes it easier to do the processing in small steps. For example, the Dome solution above first generates all the IDs and then duplicates all the questions with their IDs, whereas the XSLT calculates the ID for each question twice.

The interactive nature of Dome makes it easier to get started, since the user can start cutting and pasting right away. Even if their initial approach might not cope so well with other inputs, at least they have something to improve upon. XSLT requires more thought beforehand in order to do anything at all. On the other hand, complex problems always benefit from more thought, and a careless Dome solution quickly becomes messy if the user records more code to correct earlier mistakes rather than altering the existing code.

XSLT is harder to debug, since it often simply 'doesn't work' and the user is left trying to work out why. Since Dome programs are created by actually manipulating the data, the user always ends up with a working program after recording, at least for the example data used. When the input has a regular structure, as in this example, the Dome program will normally be completely correct first time. Dome is also better at handling variation in the input. This wasn't a problem in this

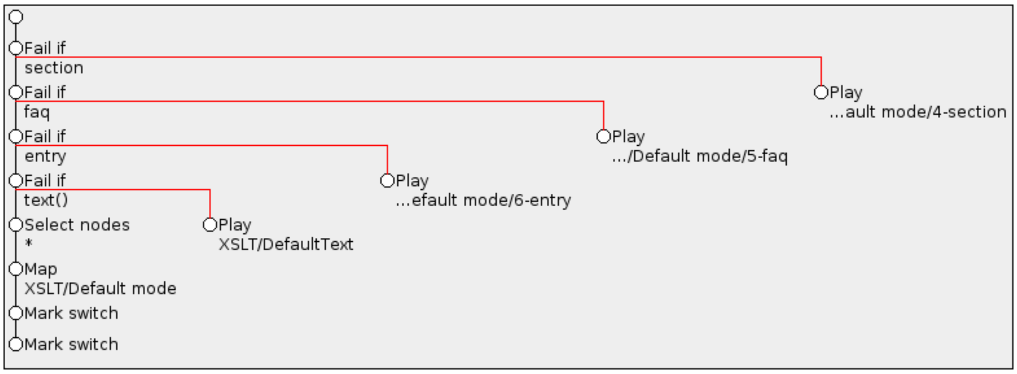


Figure 6.1: The dispatcher Dome program for the default mode.

example since the source was very structured, but the task of converting the original HTML to XML would have been very difficult in XSLT.

An XSLT transformation is easier to read and modify on its own, whereas modifying a Dome script usually requires running it on some example data.

6.3 XSLT to Dome conversion

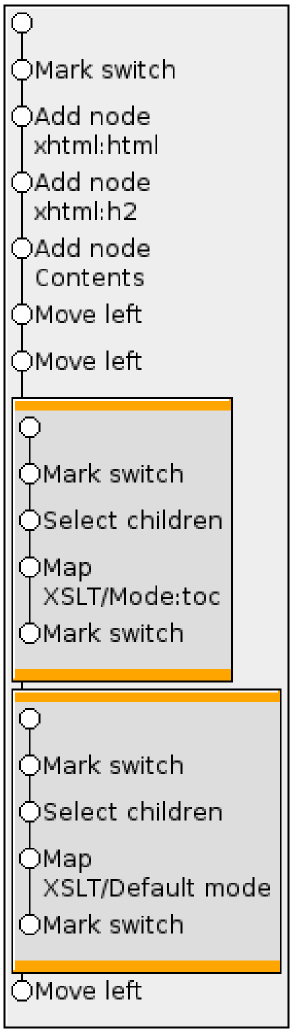
The above tasks show that Dome and XSLT are broadly similar, but differ in their typical approach. However, we can also show that Dome is able to implement XSLT style transformations. An XSLT-to-Dome converter has been written which, while not supporting all of XSLT (due to time constraints) does allow many transforms to run.

The XSLT examples above run correctly whether using Dome or a standard XSLT processor to execute them, although the Dome programs generated automatically from the XSLT take longer to run than the idiomatic Dome presented previously.

When loading an XSLT stylesheet, Dome creates two sub-nodes of the root, `<Result>` and `<Source>`. The source data file, if any, is placed under `<Source>`. Dome then constructs a program which will process the source tree to generate the result. Keeping everything in a subtree allows Dome, like XSLT itself, to produce result tree fragments (results with any number of top-level elements). `<Source>` and `<Result>` represent the *Document nodes* of the two documents, with the *Document element node* (for a well-formed XML source or result) inside.

The Root program puts the mark (a secondary cursor) on the `<Result>` node and selects `<Source>`. There is one sub-program of Root for each XSLT mode, and each one acts as a dispatcher. It finds a template which matches the selected node and runs the Dome program for that template. Therefore, the Root program starts the transformation by running the automatically-generated dispatcher program for the default mode, shown in figure 6.1, with the source's document element selected.

Each of the 'Fail if' tests corresponds to a '`match`' attribute on an XSLT `<template>`. If the template matches, the horizontal 'fail' arc is taken, causing the program for that template to be executed. If no template matches, all the child nodes are selected and the dispatcher is run recursively on each one, implementing the behaviour of XSLT's built-in default template.



The Root program has one subprogram for each XSLT mode, and each of these programs has one subprogram for each template in that mode, such as the program for the template which processes the `<faq>` element, shown on the right.

The structure of the XSLT is still visible: the 'Contents' heading is created and then all sections of the source are processed twice, first in 'toc' mode and then in the default mode.

When a template's program is run, the source node is selected and the output node to append children to is marked. The program starts by swapping the mark and cursor. The rest of the program is built up from the XSLT template recursively, starting with all the child nodes of the `<xsl:template>`:

Literal elements record a sequence which:

1. Appends a new child element with the same name (the cursor is moved to the new element).
2. Adds each attribute, evaluating an XPath expression for each value[4].
3. Recursively calls the generator on all the child nodes of the XSLT literal element.
4. Moves the cursor to the parent (restoring it to its old location).

Literal text nodes just append a new text node and move back to the parent.

value-of nodes switch the cursor and mark, evaluate the expression, switch the cursor and mark back and append the result (on the clipboard) to the output node. Finally, the cursor is moved back to the original parent node.

copy-element does a shallow copy, and recursively processes all its child nodes.

apply-templates records a new 'mark restore' block. In this, it switches the mark and cursor and evaluates the '`select`' XPath to select the nodes in the source

[4]The mark and cursor are switched around for this so that evaluation happens in the context of the source node.

document to be processed. It then loops over them, running the dispatcher program for the appropriate mode, before finally switching the mark and cursor back. A restore block restores the mark on exit to the nodes which were marked on entry. This is needed to ensure the source node is still selected if there are more operations after the apply-templates.

Support for other XSLT elements such as `<xsl:choose>` and `<xsl:if>` has not been implemented, but these conditional constructs could easily be supported using blocks and the 'Fail if' operation used in the dispatcher programs (the process of selecting the correct template to instantiate is essentially nothing more than an implicit `<xsl:choose>`).

6.4 Conclusions

In this chapter we have given an introduction to XSLT, the Web Consortium's preferred XML transformation language, which was developed originally as a stylesheet language to replace DSSSL and similar systems.

Using a real-world example of converting an XML FAQ to XHTML[5], we have compared idiomatic solutions to the problem using Dome and XSLT. We have seen that they can both be used for such tasks, but typically take different approaches. An XSLT solution to such a problem involves describing how to create a new output document by taking values from the input, whilst a typical Dome solution has the user modifying the input document in-place, turning it into the final output.

However, Dome is also able to support XSLT's before-and-after style of transformation. By building an XSLT-to-Dome conversion tool, we have established a close equivalence between the two, and have shown how a selection of XSLT constructs can be automatically converted into Dome programs.

We believe that using Dome for the kind of task presented here is considerably easier than using XSLT. Intermediate computer users who would be unable to use XSLT would be able to use Dome for the same task, and even expert users will likely find that creating a solution using Dome is quicker than using XSLT.

[5]The FAQ in question was for the ROX desktop project. The original HTML was converted to XML using Dome, then both Dome and XSLT transformations were created to convert to XHTML. The FAQ has since been migrated to a Wiki which uses a simple non-XML markup — this conversion from XML to the Wiki format was also performed using Dome.

Chapter 7

Dome as a visual language

Today's highly visual user interfaces have made computers accessible to a wide range of people. However, these interfaces typically restrict their users to a small set of built-in operations, and to recover the flexibility of a full programming language requires a paradigm shift to traditional textual programming. Visual programming aims to combine the flexibility of textual programming languages with the ease of use and familiarity of today's graphical interfaces.

This chapter provides a look at the field of visual programming, and examines some previous visual languages. We describe our initial work in this area, which led to the development of Dome, introduced in chapter 4, and consider how Dome performs as a visual language.

Visual programming[28] can be defined in many ways. Systems which use graphics to aid construction of a textual program are called *visual environments*, while languages making use of visual expressions are called *visual languages*. Both can be considered as being aspects of visual programming. *Visual Programming: Perspectives and Approaches*[121] differentiates between these two by saying that visual environments use visual techniques to *show* the user something, whereas a visual language is used to *tell* the system what to do.

The Free On-line Dictionary of Computing[62] defines a visual programming language (*VPL*) as:

> "Any programming language that allows the user to specify a program in a two-(or more)-dimensional way. Conventional textual languages are not considered two-dimensional since the compiler or interpreter processes them as one-dimensional streams of characters. A VPL allows programming with visual expressions — spatial arrangements of textual and graphical symbols."

Many visual languages employ some form of *programming by demonstration* (or 'by example'). These systems allow the user to manipulate some sample data to show the system what to do. Such systems can be divided into 'intelligent' systems, which make sophisticated, and therefore unpredictable, inferences and then interact with the user to check their guesses, and predictable systems which use fixed rules, of varying complexity, to generate a program from one or more examples.

As we have seen, Dome is a predictable non-learning programming by demonstration system. Because the user edits a Dome program through a control flow diagram, Dome can be considered a visual language. However, Dome's use of text rather than icons for each operation, and its use of structured blocks, mean that there is a natural textual representation of any Dome program.

7.1 Other visual languages

In the following sections, we describe a selection of existing visual languages to give an idea of the range of visual languages available.

7.1.1 Pygmalion

One of the oldest visual languages is *Pygmalion*[41]. Pygmalion is an 'electronic blackboard' in which formulae are written on the screen. Once all of a formula's values have been filled in, it is replaced by the answer. For example an 'add' operation is displayed as an icon containing two boxes with a plus sign between them. Once values have been dragged into both boxes, the whole icon changes into a simple number icon, showing the result. This number icon can then be dragged to other boxes, and so on. The actions used to generate the final state of the blackboard are recorded and can be replayed using different starting values.

> "One starts with an image representing the initial state and transforms each image into the next by editing it to produce a new image. The programmer continues to edit until the desired picture appears. When one watches a program execute, it is similar to watching a movie. The difference is that, depending on the inputs, Pygmalion movies may change every time they are played."
>
> — David Canfield Smith, Pygmalion's author[41]

Of course, this description also perfectly describes Dome. Smith laments that *"One feature that I didn't implement but wish I had is the ability to play movies both backward and forward; then one could step a program backward from a bug to find where it went wrong."*. Dome provides unlimited undo and redo buffers, which do indeed greatly assist in debugging.

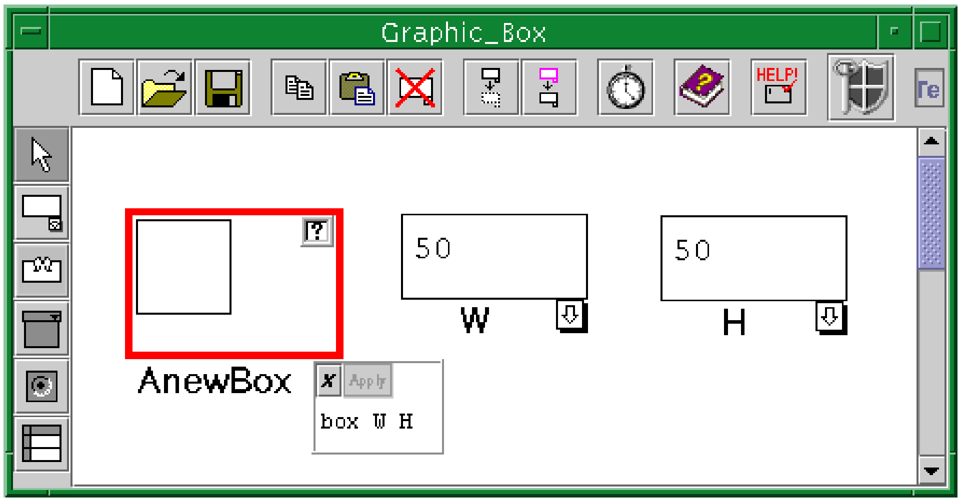


Figure 7.1: A Forms/3 form.

One limitation of Pygmalion is that conditionals must be specified as the programs are recorded. For example, when defining the factorial function, the first step is to test whether the input is 1, even though the example input on the first iteration is different. Dome can automatically enter record mode to handle new cases when it detects an error, but there are still cases where the user must plan ahead (as shown below in section 7.3.2).

7.1.2 Spreadsheets

Undoubtedly the most famous and successful of all visual languages, the spreadsheet presents a grid of cells, each of which displays the result of some calculation specified by the user. These calculations can depend on the values shown in other cells, allowing complex calculations to be broken down into separate visible steps.

Like most successful visual languages, the spreadsheet still relies heavily on text, which it uses for the formula in each cell and for the results. However, this textual part of the interface is integrated with the visual side in most spreadsheets; clicking on a cell will add its name to the current formula, and placing the cursor in a cell reference in a formula highlights the cell it refers to.

7.1.3 Forms/3

Forms/3[27] is a visual language which extends the spreadsheet concept. Cells are not fixed in a rigid grid layout, but can be positioned freely, and cell values are not restricted to numbers and strings; a formula may evaluate to a graphical result such as a rectangle or circle. The form in figure 7.1, taken from *A Guided Tour Of Forms/3*[26], shows three cells. The left-most cell, which has its formula displayed below it, contains a box which takes its dimensions from the other two cells.

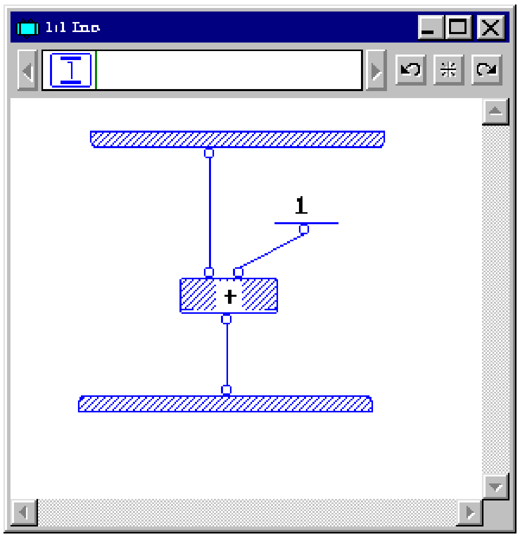


Figure 7.2: A Prograph function which increments its input argument.

User defined types can be created in Forms/3 by creating a new form (sheet) and marking some of the boxes as private (these will be hidden when the type is later used), leaving the rest as the public interface. Instances of the type are created by cloning the form. The cells in the clones remain linked to the original unless they are changed manually, in the same way that an instance of a class in an object oriented language will use its class's current setting for an attribute until a value is assigned directly to the instance.

Forms/3 also has a concept of *logical time*, including support for event handling. One interesting effect of time support is that it allows algorithms written in Forms/3 to be animated by additional Forms/3 code, as described in *Integrating Algorithm Animation into a Declarative Visual Programming Language*[29].

A key feature of the language is that, unlike most other spreadsheets, the extensions it provides do not depart from the formula based approach. Other spreadsheets typically provide a traditional programming language for more complex operations.

Forms/3 is a very flexible programming language which does not require users to move to a significantly different mode of operation to access its more advanced features, although techniques such as cloning forms to implement recursion will still be beyond most users.

7.1.4 Prograph

Prograph[107] is a commercial visual language in which the programmer draws data-flow diagrams to create programs. As with textual programming languages, the program is visible, but the data must be imagined (though it is possible to examine the data at any point in a program as it executes; this is similar to using a debugger in a textual language).

When defining a function in Prograph, the input arguments appear as connection points on a bar at the top of the function, and the outputs are on a bar at the bottom. Other functions called from within the function are shown as boxes with inputs on the top and outputs on the bottom. By connecting these terminals together, a visual program is formed. Figure 7.2 shows a Prograph function which increments its input and returns the new value.

Although Prograph is essentially a functional language, functions can have side-effects and there is a separate visual notation to force a particular order of execution within a function. As in many visual languages, Prograph programs can be edited and run at the same time. This allows the 'programming by debugging' programming style to be used, in which parts of the program are written (and checked) as they are needed during execution. This saves the programmer from having to think ahead too far, and ensures that a concrete example of the data is available as the function is written.

Prograph is somewhat unusual among visual languages in that it has been used for some large real-world programming tasks, while still being a fully visual general purpose language. Most visual languages are either highly domain specific, or limited to toy problems.

In *Using visual programming to extend the power of spreadsheet*[40], a Prograph program is used instead of a textual formula for each cell in a spreadsheet. Most of the advantages of the proposed system (such as specifying which cell references are fixed when a formula is copied rather than when it is created) are not directly due to the use of Prograph, however, but would be useful in traditional spreadsheets too.

7.1.5 PSh

PSh[52] is a visual shell implemented in Prograph. Several of the standard Unix commands have been reimplemented in Prograph, often taking advantage of the new syntax. For example, the 'wc' (word count) command (shown in figure 7.3) now has three separate output terminals for lines, words and characters rather than using input flags to set the desired output format.

Part of the challenge for PSh is that, while it is a graphical tool, its style is very different to normal direct manipulation editors with which users are familiar. For example, a user would normally count the number of words in a file by choosing 'Word count' from a pull-down menu, not by connecting the document's icon to a word-count function and connecting the output from that to a display function.

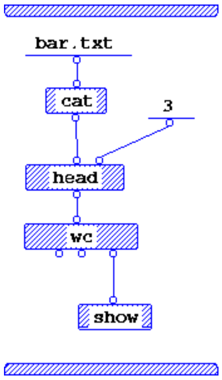


Figure 7.3: PSh command to count the number of characters in the first three lines of a file.

Therefore, other applications must adopt a PSh-style interface themselves for non-programming tasks before it can really bridge the gap between programming and non-programming computer use.

7.1.6 Cocoa

Cocoa[123] (previously known as *KidSim*) is a visual language intended for children. The current state is shown as a grid of tiles and the state is updated by visual search-and-replace rules which look for a particular pattern of tiles (for example, two tiles showing a person facing a wall) and specify a replacement set (for example, a person facing away from the wall; this rule indicates that when a person reaches a dead-end, they will turn around).

Because Cocoa is designed for children, it has a number of features which make it easy to use. It shows the current state of the system visually at all times; there are no hidden values in the system, so the behaviour is entirely predictable from the current state of the grid and the list of rules. It has no complex control structures, just a list from which the first match is taken. Although general purpose visual languages need to be more complicated than Cocoa, it does show that visual languages can be made extremely easy to use.

Altaira[106] is a similar system, designed for controlling robots. It is slightly more complicated than Cocoa since the rules must match not only the robot's surroundings but also the state of its sensors. Also, the output is not directly a new tile state (since this is a real robot, not a virtual world) but updates to the stored tile map and motor outputs. Altaira demonstrates that the Cocoa interface can be extended to more advanced problems.

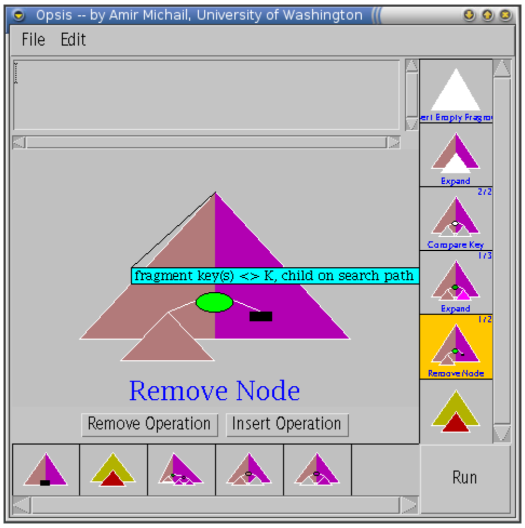


Figure 7.4: An Opsis abstract tree.

7.1.7 Opsis

Opsis[92] is an educational tool which allows users to describe binary tree algorithms using a visual notation. The user builds a graph of *abstract states* (abstract states are concepts such as 'a tree containing a node K', or 'a subtree whose root node is greater than K'). Part of a state can be *expanded* to create several possible sub-states. For example, expanding a part representing 'any subtree' creates two arcs out of the state to new abstract states, one with the part replaced by 'an empty subtree' and in the other with 'a node with left and right subtrees'.

Figure 7.4 shows an abstract tree containing a node matching the search target node, which has an empty right subtree, and may or may not have an empty left subtree. The vertical list of states shows the path from the initial (most general) tree to the current particular case, while the horizontal list shows all the final states (those for which no successor states have yet been defined).

Opsis is designed to teach only a very limited range of algorithms, almost forcing the 'correct' program to be created. Its domain is searching, inserting and deleting operations on ordered binary trees. Opsis is typical of many tree and graph based visual languages in that it focuses on very formal and abstract operations and is unsuitable for general purpose programming. The third year computer science students who evaluated the Opsis system reported that it had "a non-trivial learning curve".

7.2 Our previous work

Before starting work on Dome, we had created two previous visual languages. The first, VVSh, was a visual Unix shell which displayed commands as processes

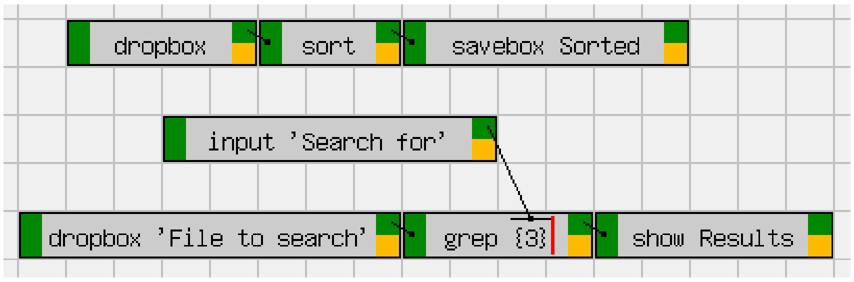


Figure 7.5: Two VVSh pipelines. The top one sorts a file dropped on it; the lower one will search a text file for a string.

connected by links, representing pipes. VVSh was not intended to be a general purpose programming language, but was designed for connecting small numbers of components together. The second system was based around multiple threads, with two-way communication between them.

7.2.1 The Very Visual Shell

VVSh (the *Very Visual Shell*) is a graphical version of the Unix shell interface. Unlike PSh (see section 7.1.4), which requires all existing commands to be rewritten in Prograph, VVSh components are unmodified Unix commands.

Each command is shown in a box, and the boxes are connected by lines (representing Unix pipes). The idea was to keep the flexibility of the command-line but with an easier syntax. The two-dimensional nature of the display actually allows more flexibility than the command-line. For example, two pipelines can feed into a single command (see figure 7.5).

Each command has three connectors, corresponding to Unix's three special file descriptors: *standard input*, *standard output* and *standard error*. The textual output from a command can also be inserted (*injected* in Prograph terminology) into the text of another command; the `grep` command in the example above receives the string to search for in this way.

VVSh allows commands to be typed in as they would be in a shell and automatically creates the visual pipeline as this is done, so expert users can enter commands as quickly with VVSh as with a traditional shell. VVSh's main weaknesses as a general purpose language are that it scales poorly, and that looping is cumbersome.

Since each box is a Unix process, having too many is very inefficient. VVSh has no support for modules or nested structures beyond having a box run another VVSh program. This makes it unsuitable for large projects as everything must appear on the same sheet.

Looping is difficult because every box is executed exactly once each time a program is run. Therefore, VVSh could not be used to create a loop directly. Although it is

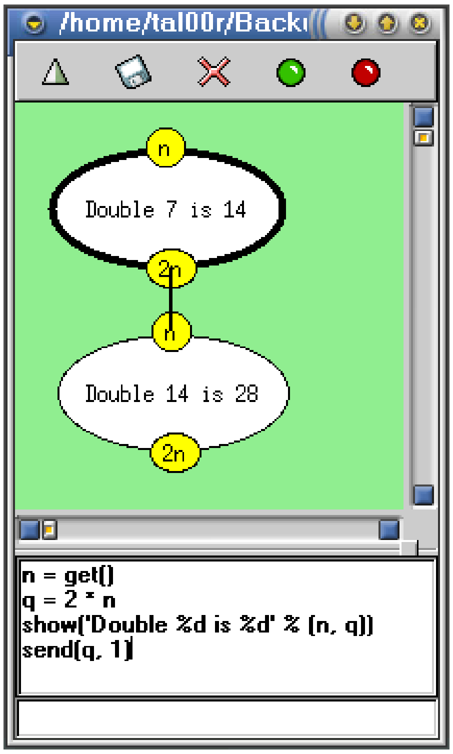


Figure 7.6: Two threads, each with the same code. Entering a number in the top port will double it twice and send the result out of the lower port. Each thread can display a message in its ellipse showing what it is doing.

possible to use '`xargs -n 1 script.vv`' to run another whole VVSh script once for each line of input, this syntax is clumsy.

7.2.2 Communicating threads

There are a number of styles of visual language. Some are based on control-flow (flowcharts, for example); others are based on data-flow (like VVSh). There are two main types of data-flow visual languages around: *data-push* (in which the output of one block triggers the next block to run) and *data-pull* (in which a block is run when its output is required by another block).

Our attempt to generalise these existing systems consists of a collection of threads (each represented by an ellipse) which each have a number of bi-directional ports. To construct a program, these ports are joined together. The system is capable of data-push and data-pull.

Each of the two identical threads shown in figure 7.6 reads a number from its top port, doubles it, and writes the result to its lower port. By connecting the output from one thread to the input of the next, a system that multiplies by four is created. The code for the individual threads had to be written in python — the idea was to build a library of code for novices to use.

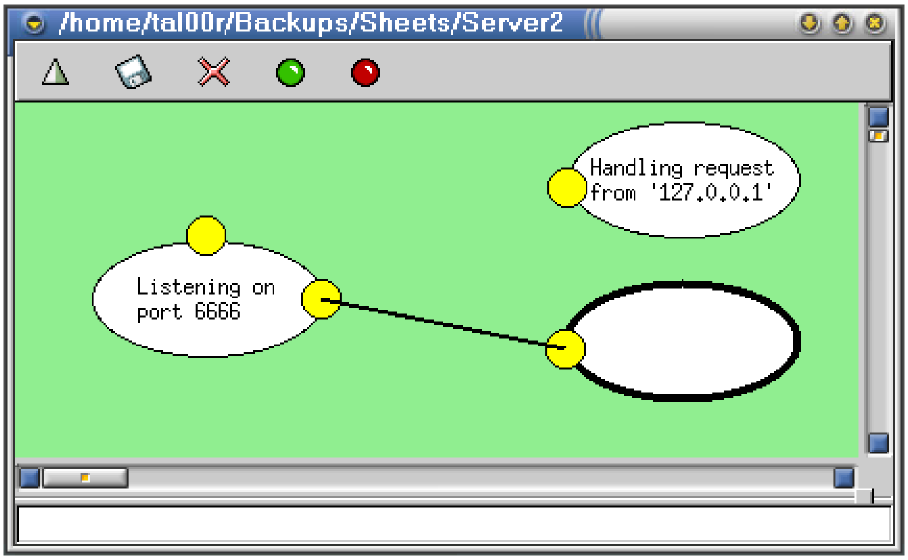


Figure 7.7: A visual server. The ellipse (thread) on the left listens for TCP connections and creates a new copy of the blank ellipse to handle each connection. The program is currently dealing with one connection while listening for a second one.

The system worked well provided the number of threads was fixed. For many systems, such as Internet servers, new threads must be created as the program runs. This is possible, but there is the risk of the layout becoming confused as the system has to place new threads itself (see figure 7.7).

The design allows for nested programs, although this has not been implemented, but requiring a separate thread for each operation makes it infeasible to use too many anyway. The main problems with this prototype were:

Resource usage Although the threads are more light-weight than the processes used in VVSh, it is still impractical to use too many at once.

Layout The user must spend time laying out all the threads manually. Automatic creation of new objects leads to messy layouts.

Invisible state In VVSh, processes have no internal state when not running, but here each thread can keep values while idle. This hidden information makes a ellipse's behaviour unpredictable just from a visual inspection.

7.3 Dome

From our experiences with these languages several conclusions can be drawn, of which the most important is that a visual language should make all the data in the system explicit. There should be no hidden state, as this makes understanding the operation of the system difficult.

The need to represent state visually has been a common theme in easy-to-use programming languages, such as *Rehearsal World*[41] with its *off-stage performers*, and the *hidden controls* in *Visual Basic* (which is considered to be a visual environment, rather than a visual programming language, since the language itself is textual).

> "Since data structures and database schemas play a prominent role in programming, and since graphical depictions found in every good textbook are known to be extremely helpful, it is natural that visualisation of data structures and database schemas is pursued."
>
> *Visual Programming: Perspectives and Approaches*[121]

The previous systems were difficult to use in part because the data they were processing was not visible. All the data in Dome is shown as a tree, and this solves both the visualisation and automatic layout problems. Further, by having an extended range of operations available, the user is able to specify their meaning precisely. This allows Dome to be used as a programming language without departing too far from the familiar direct-manipulation interface style.

A tree is excellent for storing many types of data, is easy to lay out automatically (unlike a graph), and is scalable (a subtree can be collapsed and treated as a leaf, or a number of subtrees can be sent to different processes to be dealt with in parallel).

In previous chapters, we have seen Dome as a transformation language for external XML documents. In the following sections, we show how Dome be used in very different ways, such as for algorithm visualisation, but where its visual data structures and record-and-playback programming model still make program creation easier for non-expert programmers.

7.3.1 Noughts and crosses

Dome can render a subtree as a vector image. This was used to implement a simple graphical noughts-and-crosses game. The program changes the DOM tree, and the changes are automatically reflected in the graphical view. HTML can be viewed in a similar way, and we could therefore replace the current non-standard system with SVG.

In figure 7.8, the X and O nodes are collapsed group nodes; the actual drawing primitives are inside. Note also the use of `<tr>` nodes to force a horizontal layout, allowing a close mapping from the internal program representation to the display[1]. Nodes can use an attribute to specify a program to run when the graphical representation is clicked, similar to HTML's '`onClick`' attribute.

[1] The early version of Dome used for this had special formatting rules for HTML's table row element. This feature is not available in the current version, although it is hoped that it can be implemented with a more flexible scheme based on stylesheets.

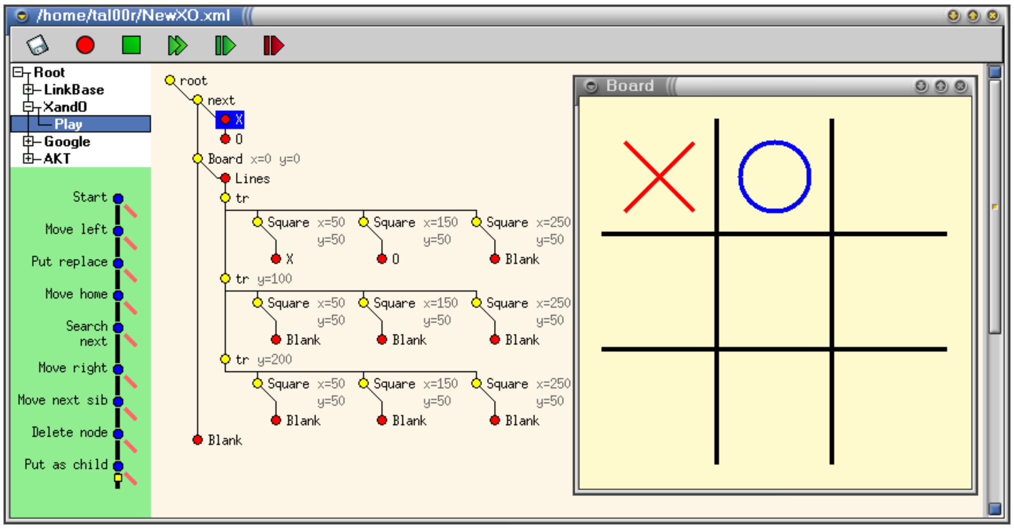


Figure 7.8: Dome playing noughts and crosses.

7.3.2 Algorithm visualisation

Figure 7.9 shows a Dome implementation of the *quicksort* algorithm traditionally used by visual language researchers as a test problem. The user begins creating the program by selecting the first `<item>` and renaming it to `<pivot>`. They then create a `<low>` element before it and a `<high>` element after it.

The remaining items are selected and a foreach block created to loop over them. In the block, the user records an assertion (an XPath) that compares the current node's value with the pivot (for example, `number(.)<number(../pivot)` if the first item is smaller than the pivot), and drags the item to the appropriate group (`<low>` or `<high>`).

The user clicks Play, causing the loop to continue until the assertion fails, and drags the selected item to the other group. The loop is resumed again and this time runs to completion, partitioning all the items into the two groups.

The two groups are then selected and an enter–leave–foreach block created to process them. In this block, the qsort program is run recursively. During this recursive execution, the base case will be reached (an empty list) and recording starts on the initial select item operation. The user records a 'pass' operation to simply return (since an empty list is already sorted), and continues execution.

After the deepest recursive call returns, recording resumes after the loop block, with the `<high>` and `<low>` elements selected. These are removed, and the pivot renamed back to `<item>`, leaving a sorted list. The program is now run to completion.

There is one point in this sequence where care must be taken. After the initial partitioning step, both `<low>` and `<high>` are normally selected. However, if no

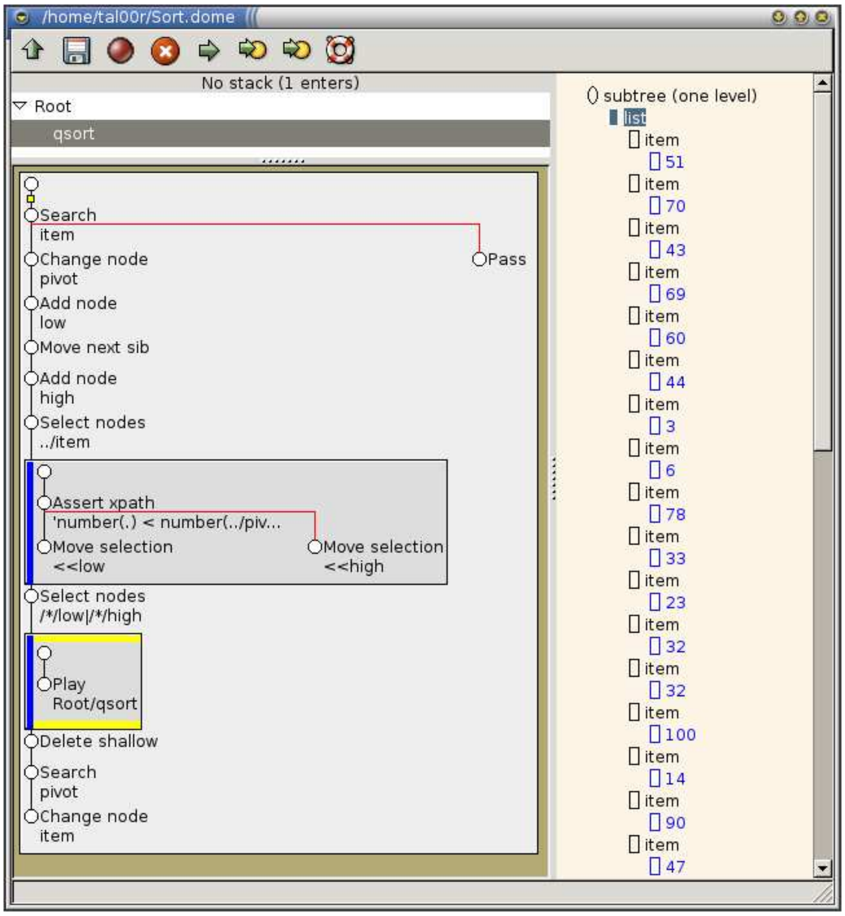


Figure 7.9: An implementation of the quicksort algorithm in Dome.

items were moved to a group then that group will not be selected. Therefore, we must manually select them anyway, even though this may appear to be unnecessary when recording. The problem is easily discovered using Dome's debugging features, such as single stepping, breakpoints and the ability to undo and redo any changes to isolate a problem.

A faster alternative to looping over all the items with an assertion to partition them is to write an XPath to select all items less than the pivot and move them all directly. However, the approach presented here is more suitable for demonstration of the quicksort algorithm, since it makes the loop explicit.

7.4 Conclusions

In this chapter we have provided an introduction to the field of visual languages, and in particular to the spreadsheet and some of the many attempts to extend it. We have looked at the Forms/3 language, which generalises the spreadsheet from a grid layout to any collection of cells, and at work which replaces the spreadsheet's textual formulae with visual Prograph expressions. We have also seen that Pygmalion shares Dome's approach of editing an initial state into a final state and recording the operations for later application to other input data.

We have presented our two earlier efforts to create a visual language, and shown how these experiences also influenced the development of Dome, in particular by highlighting the need for visual data-structures. The earliest of these, VVSh, was purely a scripting tool, used to 'glue' different programs together. The second one used threads, linking smaller components together but still requiring textual programing for the individual pieces. Dome is a full programming language, capable of performing a wide variety of tasks without recourse to another language.

We then described a number of uses for Dome which exploit its power as a general purpose programming language, rather than as a transformation language for external XML data, which has been covered in previous chapters. We have seen that, despite having no intelligent learning algorithms, programs such as 'quicksort' can be implemented quite naturally by demonstration in Dome.

We have also shown that a Dome program can provide a fully visual custom interface, using a canvas widget, and that the internal state information can be designed to follow closely the graphical output, making the program easier to understand. Although the implementation shown uses a custom format, a selection of SVG and HTML rendering components are now available, which could be supported by Dome to give a great deal of flexibility in the interfaces produced. It would also be possible to support stylesheets within Dome itself, allowing the main display area to be customised for any particular application (such as displaying all `<title>` elements in bold).

Chapter 8

MIME database

While Dome, as a generic XML editor, treats all XML-based formats equally, it must also cope with many other types of data. When used for web harvesting (as described in chapter 5), for example, Dome must recognise HTML files and convert them to XHTML that it can read. When loading a plain text file, Dome should try to import it as a single block of text, rather than by trying to parse it as HTML or XML. Dome could also use import filters to convert various other formats, such as *CSV* (*Comma Separated Values*), to XML that it can edit directly. All this requires a system to recognise the types of files and to discover information about each type.

The *MIME*[14] (*Multipurpose Internet Mail Extensions*) system is a naming scheme for file types. In this chapter we consider various systems for mapping a file's name or contents to its MIME type. We propose an XML based format to replace the existing methods for specifying this mapping, and describe some of the software written to use the new scheme. We take advantage of the extensible nature of XML to allow information *about* each MIME type to be specified in the same place, yet accessed efficiently.

Efficiency is particularly important for small utilities (such as the hypothetical *any2xml* importer utility suggested above), which may be invoked many times during processing. Start-up time is also an important consideration for interactive programs, where reading all the source XML for the MIME definitions can add significantly to the start-up time of an application.

A MIME type consists of a *media type* and a *subtype*. The media type gives the general category (such as text, image or audio) and the subtype gives the specific format. Some examples of MIME types are 'text/plain', 'text/html', 'text/xml' and 'image/png'. Applications such as Dome can use the MIME type of some data to decide how to process it. For example, ‘text/html’ data needs to be converted to ‘text/xml’, whereas ‘text/plain’ should be loaded as a single text region.

MIME types are used in many Internet protocols, such as HTTP (used to fetch web pages), and to give types to email attachments. Many programs and desktop environments also use the MIME system to represent the types of files. This is used to choose a suitable icon for each file and to decide which applications can be used to open it. When using drag-and-drop between applications, the MIME type is used by the receiver to specify the format in which it would like the data, if the sender offers multiple formats.

Since most filesystems do not provide a way to store the MIME type of a file explicitly, and since most applications don’t set such an attribute even if available, there needs to be another mechanism to determine the MIME type for any given file. This is generally done by examining the file’s name or contents, and looking up the correct MIME type in a database. The database might state that files with names ending in ‘.png’ have the MIME type ‘image/png’, or that files containing ‘`<HTML>`’ near the start have the type ‘text/html’.

In this chapter we start with a brief overview of the string matching systems used by many of the MIME schemes for recognising files. We then provide summaries of the most common existing MIME databases, and discuss some of their shortcomings. Our proposed replacement system is discussed next, showing how it overcomes these problems. We end the chapter with a look at the state of the proposal’s adoption.

8.1 Glob patterns and regular expressions

There are two pattern matching systems used for recognising filenames: *glob patterns* and *regular expressions*. This section provides a very brief introduction for readers who are not familiar with them.

Regular expressions were first used for text matching by Ken Thompson, as described in his 1968 paper[128], though they were originally invented by Stephen Kleene for modelling neural networks (matching particular input sensations which should cause a response).

The simplest regular expression is a single character, which matches that character in the filename. So, the regular expression ‘a’ will match the string ‘a’. The ‘.’ character matches any character, so ‘.at’ matches ‘Bat’, ‘Cat’, ‘Hat’ and so on. ‘*’ means that the proceeding item can match zero or more times, so that ‘a*’ matches ‘’, ‘a’, ‘aa’, ‘aaa’ and so on and ‘.*’ matches anything or nothing. ‘`^`’ matches only at the start of the string, while ‘`$`’ only matches at the end. Subexpressions may be grouped by parenthesis, and '|' allows either of two patterns to match. For example, '(zero|one)*' matches any interleaving of 'zero' and 'one' strings. To match any character in a set, brackets may be used, as in 'index-[a-z]' to match 'index-a', 'index-b', and so on. If the '^' character appears at the start of the set then the expression matches any character *not* in the set.

Glob patterns[1] are simpler but less expressive than regular expressions. In a glob pattern, '?' is equivalent to '.' (any character) and '*' to '.*'. Bracketed ranges can also be used, as described above. No other features of regular expressions are available. Glob patterns must match the entire filename, so start and end markers are not needed.

In both types of pattern, a '\' character disables the special meaning of the following character. There is some variation between regular expression engines as to whether certain special characters have their special meaning only when escaped, or only when not escaped. For comparison, the following patterns are equivalent:

Glob pattern	Regular expression	Example match
`*.html`	`^.*\.html$`	index.html

8.2 Other systems

In the past, Linux and Unix systems have had many independent MIME databases, such as those provided by:

- The GNOME desktop[53]
- The KDE desktop[73]
- The ROX desktop[82]
- The Debian[43] distribution's mime-support package
- The Apache web server[3]
- The Unix 'file' command[148]

This was extremely confusing for users, as installing a new package typically only updated one of these databases, and when an application got the type wrong, the user had to find an editing tool for whichever database that application used.

Also, the MIME specification only defines a small number of types, and each of the database maintainers had made up different MIME types for unregistered types in a large number of cases. A typical example being the use of 'text/x-tex' in one database, and 'application/x-tex' in another, both referring to TeX files. This meant that drag-and-drop between different programs was unreliable, and

[1] "*glob* was a subprogram that expanded wild cards in archaic pre-Bourne versions of the Unix shell" — The Free On-line Dictionary of Computing.

that it was not possible to share icon themes between applications using different databases.

In the following sections, we take a brief look at each of the schemes listed above. Since all the contents matching systems are based on 'file', it is helpful to understand that section (8.2.1) in particular.

At the time of writing, ROX has been converted to the new system, while KDE and GNOME are in the process of moving over. The 'file' and Apache projects have not started conversion, or even decided whether they will convert. However, for consistency, we refer to all the systems in the past tense.

8.2.1 The Unix 'file' command

This program attempts to identify files by their contents only. It has a single file containing all the rules, named '`magic.mime`'. Each line contains an offset, a type, a value and a MIME type. If the file contains the given value at the given offset then it is of the given MIME type.

If a line begins with one or more `>` characters then it is a *continuation line*. Both it and its parent (the previous line with one less `>` character) must match.

Example file:

```
0       string          !<arch>          application/x-archive
>8      string          debian           application/x-debian-package
```

This specifies that if a file begins with the string '`!<arch>`' then it has the MIME type 'application/x-archive', unless it also contains the string 'debian' at offset 8, in which case it is 'application/x-debian-package'.

The 'file' command is also able to provide a textual description of a file's type, often including some information from the file, such as the width and height of an image.

8.2.2 The KDE desktop

KDE[73] (the *K Desktop Environment*) used a format based on the old Windows '.ini' format, in which a file is divided into a number of sections each having a number of $(key, value)$ pairs. There was one file for each MIME type, named after the type. Each file contained a 'Desktop Entry' section with the 'Type' key set to 'MimeType'. The files were arranged in the filesystem to mirror the two-level MIME type hierarchy.

Example file, named '`application/x-kword`':

```
[Desktop Entry]
Encoding=UTF-8
MimeType=application/x-kword
Comment=KWord
Comment[af]=kword
[... etc. other translations ]
Icon=kword
Type=MimeType
Patterns=*.kwd;*.kwt;
X-KDE-AutoEmbed=false

[Property::X-KDE-NativeExtension]
Type=QString
Value=.kwd
```

These files only specified rules for guessing the type from the file name, using the 'Patterns' fields to specify a list of glob patterns, any of which could match. The patterns were case-sensitive.

A single extra file stored all the rules for recognising files by content. This is almost identical to *file*'s '`magic.mime`' database file, except for the optional encoding field, which is missing from KDE's version.

The format is described in the file itself as follows:

```
# The format is 4-5 columns:
#    Column #1: byte number to begin checking from, ">" indicates continuation
#    Column #2: type of data to match
#    Column #3: contents of data to match
#    Column #4: MIME type of result
```

8.2.3 The GNOME desktop

GNOME[53] (the *GNU Network Object Model Environment*) uses the *gnome-vfs* library to determine the MIME type of a file. This library loads name-to-type rules from files with a '.mime' extension in a system-wide directory (set at install time), and merged with those in the user's directory. It loads textual descriptions for the types (that is, information *about* the type, not information for *determining* the type) from files in the same directories, ending with '.keys'. The file '`gnome-vfs.mime`' in the system directory is always loaded first, allowing everything else to override it. The file '`user.mime`' in the user's directory is always loaded last, making these settings take precedence over all others.

The format of the .mime files are described as follows (extract from the main '`gnome.mime`' file):

```
# The format of this file is:
#
# mime-type
# ext[,prio]: list of extensions for this mime-type
# regex[,prio]: a regular expression that matches the filename
#
# more than one ext: and regex: fields can be present.
#
# prio is the priority for the match, the default is 1. This is required
# to distinguish composed filenames, for example .gz has a priority of 1
# and .tar.gz has a priority of 2 (thus a file having the filename
# something.tar.gz will match the mime-type for tar.gz before the mime-type
# for .gz
```

When looking up the type for a file, gnome-vfs looks first for an exact-case match for the extension, then an all upper-case match, then an all lower-case match. If no matches are found, or there is no '.' in the name, then the regular expression matches are checked. It does both steps first for rules with priority 2, then for those with priority 1. The modification time on the '`mime-info`' directories is used to detect changes.

The '.keys' files contain type-to-description rules, as shown in this extract:

```
application/msword
        description=Microsoft Word document
        [de]description=Microsoft Word-Dokument
        ...
```

The format for magic entries is, as with KDE, based on the 'file' syntax. All the rules are stored in the '`gnome-vfs-mime-magic`' file within 'SYSCONFDIR' (set when GNOME is installed). Continuation lines are not supported, but it adds ranges:

```
# The format of magic entries is:
#
#     offset_start[:offset_end] pattern_type pattern [&pattern_mask] type
#
# <offset_start> and <offset_end> are decimal numbers (file offsets).
#
# <pattern_type> is (byte | short | long | string | date | beshort |
#                    belong | bedate | leshort | lelong | ledate).
#
# <pattern> is an ASCII string with non-printable characters escaped
# as hex or octal escape sequences, and spaces and other important
# whitespace escaped with '\'.
#
# <pattern_mask> is a string of hex digits. The mask must be the same
```

```
# length as the pattern.
#
# <type> is a valid MIME type.
#
# Order magic patterns such that ambiguous ones (such as
# application/x-ms-dos-executable) are at the end of the list and
# therefore get applied last.
```

8.2.4 The ROX desktop

We have now modified *ROX*[82] (*RISC OS on X*) to use the new system proposed in this chapter. This section describes the previous implementation. This author is also the main author or ROX, but that work does not form part of this thesis.

ROX searched all the '`MIME-info`' directories in 'CHOICESPATH'[85] ('`~/Choices/MIME-info:/usr/local/share/Choices/MIME-info:/usr/share/Choices/MIME-info`' by default). Files from earlier directories overrode those in later ones, but the order within a directory was not specified. The file format was based on GNOME's, with the following changes:

- There were no .keys files, so files of all extensions were loaded.
- The priority was ignored.
- A case-sensitive match was tried first, then a lower-case match. No upper-case match was tried.
- Multi-part extensions were allowed. For example:

  ```
  application/x-compressed-postscript
  ext: ps.gz eps.gz
  ```

When looking up the type for a file, ROX started with the first '.' and tried a case-sensitive match of the remaining text against the extensions. It then tried again with the filename in lower-case, then began again from the second '.', and so on. The effect of this was that the longest extension match would be used, so there was no need for GNOME's priority system.

Upper-case matching was felt to be unnecessary since case-sensitive extensions are very rare, and the few known ones are covered by the exact-match and lower-case match rules alone. The upper-case match step would only be useful in the situation where the case is important, but can be anything except all lowercase.

If no type was found based on the extension, ROX tried the regular expressions. ROX had no rules for determining a file's type from its contents, nor for providing a description of a type in plain English (or any other language).

8.2.5 Debian's mime-support package

The *Debian GNU/Linux*[43] distribution allows packages to place files in the directory '`/usr/lib/mime/packages`'. They then run the 'update-mime' command, which merges these files to create '`/etc/mailcap`'. This file is used by email clients to discover which program to use to view a particular attachment.

It also provides a single file, '`/etc/mime.types`', which is a mapping from file extension to MIME type. Each line lists a MIME type and a number of possible extensions for it, for example:

```
text/html    htm html
text/mathml  mml
```

Other packages cannot modify this mapping, and it does not allow matching by file contents or even on the whole name (only extensions).

8.2.6 The Apache web server's MIME system

Apache[3] has two modules, 'mod_mime' and 'mod_mime_magic', for determining MIME types. For matching names, only file extensions can be listed. This is done by adding lines to the `httpd.conf` configuration file, in the form:

```
AddType image/gif .gif
```

The extension argument is case-insensitive, and can be specified with or without a leading dot.

'mod_mime_magic' uses a version of the file command, modified for extra speed. It uses a magic file with the same format as file's.

8.2.7 ACAP Media Type Dataset Class

The *ACAP Media Type Dataset Class*[101] draft proposes a network-based solution to the problem. The system is intended mainly to allow email clients to find a viewer application for an attachment. The draft gives the following example of a record for the 'image/jpeg' type:

Attribute	Value
entry	JPEG image
mediatype.common.type	image/jpeg
mediatype.common.extension	jpg
mediatype.common.extensionOther	(jpeg jpe jfif jfi)
mediatype.common.description	JPEG is an image format ...
mediatype.common.suppressWarning	1
mediatype.macOS.type.bin	JPEG
mediatype.macOS.creator.bin	JVWR
mediatype.macOS.creator.name	JPEG View
mediatype.macOS.action.edit.bin	8BIM
mediatype.macOS.action.edit.name	Adobe Photoshop
mediatype.unix.action.view	xv

Only extension matching and very simple content matching are supported for determining a file's type. Finding a type involves sending a query to a remote database, which is intended to allow companies to configure MIME information on a company-wide basis.

8.2.8 Summary of existing systems

All the above systems have problems with scalability, since they do not allow third-party applications to install extra information about all the aspects they cover. All systems except 'file' support matching based on name. All systems except ROX and Debian can match on file contents.

KDE, GNOME and Apache were the systems that performed matching on both names and content. They all kept the contents and name rules separate, sometimes choosing a different MIME name to represent the same type in the two places! There was also a difference in the ordering: KDE used filename matching in preference to contents matching, whereas GNOME did it the other way around.

The ACAP based system is only a draft, and so has not been considered with the above. It is not designed for handing a large number of lookups quickly, making the system too slow for use in file managers and similar. Both its name and contents matching are more primitive than other systems; only extensions can be matched, case is ignored, and the contents matching can only match an exact string at the start of the file. It provides similar information to our proposal, except that it defines many items of platform-specific information in the core specification, instead of using namespacing to allow vendor extensions.

Filename matching

GNOME and ROX allowed applications to install additional rules for matching on names, although GNOME does not make the location of the data available to non-GNOME applications, which limits its usefulness. In the GNOME files, the priority field was only being used so that regular expression matches on names would take precedence over extension matches in order to support files with double extensions (such as '.tar.gz'; a gzip-compressed tar file). In ROX, the extension mechanism was extended to handle this case directly, giving a simpler format which is also faster to match on.

Neither GNOME nor ROX really needed the flexibility offered by regular expressions over the simpler filename globs used in KDE, and careful examination showed that most of the regular expressions that were used had basic errors.

These were the correct regular expression rules in GNOME's main database:

```
^core$
\.cpio\.gz$
\.xcf\.gz$
\.xcf\.bz2$
^RMAIL$
\.anim[1-9j]$
```

These two were missing a leading '`\.`':

```
tar\.bz2$
tar\.gz$
```

These were missing a `^` at the start or `$` at the end:

```
gmon\.out$
AUTHORS
COPYING
CREDITS
INSTALL
README.*
[.|]gtkrc
[Mm]akefile
```

All of the above could have been done with normal glob patterns, and the authors would have probably got them right. The only two which were not possible to express directly with glob patterns were these two, which are also incorrect (the authors are confused about the meaning of '.' in a bracket expression, and have tried to escape it). They are intended to match Unix man pages. The fact that the allowed values for the section part differ was also probably unintentional:

```
([^0-9]|^[^\.]*)\.([1-9][a-z]?|n)\.g?[Zz]$
(([^0-9]|^[^\.]*)\.([1-9][A-Za-z]*|n)|\.man)$
```

These use the 'choice' ('`|`') and 'optional' ('`?`') operators, which would require splitting them into several patterns, one for each possible combination. They also apply '`*`' to ranges, whereas only '`.*`' can be converted to a glob pattern. However, this would only slightly broaden the allowed set of filenames.

Contents matching

All systems for getting a MIME type from a file's contents were based on the syntax of the old Unix 'file' command. All of them required a central authority to maintain that part of the database, which prevented applications from installing rules to match their own types. Also, the location of the database was often set at install time, and thus not available to programs distributed separately.

The formats were text based, and had complex escaping rules. Perhaps because of this, all the implementations were directly based on the original 'file' utility's code.

Translations and descriptions

The 'file' utility provided the most flexible descriptions, since they could include information extracted from the file itself. 'file' and KDE both provided descriptions from a central authority, while GNOME allowed third-party extensions. ROX, Debian and Apache did not use descriptions. GNOME and KDE both allowed descriptions to be translated.

8.3 Proposed new system

In discussions about these systems, it was clear that the differences between the databases were simply a result of them being separate, and not due to any fundamental disagreements between developers. Everyone was keen to see them merged.

The new unified specification proposes:

- A standard way for applications to install new MIME related information.
- A standard way to get the MIME type of a file.
- A standard way to get information about a MIME type.
- Standard locations for all the files, and methods for resolving conflicts.

For name matching, KDE's glob system replaces the dual 'ext' and 'regex' fields used by GNOME and ROX. It is trivial to detect a pattern in the form '*.ext' and store it in an extension hash table internally, and glob patterns were judged more suitable for filename matching for the reasons given above.

Matching of file contents is a union of the GNOME and KDE systems, allowing all rules from both desktops to be expressed. Specifically, it allows both continuation lines and ranges to be used. All previous content matching systems relied on the order of the rules within the file to prioritise some rules over others. Since the new scheme must combine multiple sources, each rule can now be given a priority explicitly if the ordering is important. The priority also allows highly reliable match rules to be tried before the name matching, and the rest tried afterwards.

The new system also adds a new form of contents matching not used by any of the previous systems — matching on the namespace or local name of the root element of an XML document. This provides a very reliable way to identify different kinds of XML documents, which cannot be done correctly with simple string matching.

8.3.1 File locations

There are two important requirements for the way the MIME database is stored:

- Applications must be able to extend the database in any way when they are installed, to add both new rules for determining type, and new information about specific types.
- It must be possible to install applications in '`/usr`', '`/usr/local`' and the user's home directory (in the normal Unix way) and have the MIME information used.

This specification uses the *XDG Base Directory Specification*[7] to define the prefixes below which the database is stored. In the rest of this chapter, paths shown with the prefix '`<MIME>`' indicate the files should be loaded from the '`share/mime/`' subdirectory of every directory in `XDG_HOME:XDG_DIRS`, as required by that specification.

For example, when using the default paths, "Load all the '`<MIME>/text/html.xml`' files" means to load '`/usr/share/mime/text/html.xml`', '`/usr/local/share/mime/text/html.xml`', and '`~/.local/share/mime/text/html.xml`' (if they exist).

8.3.2 Providing MIME information

Each application that wishes to contribute to the MIME database installs a single XML file, named after the application, into one of the three '`<MIME>/packages/`' directories (depending on where the user requested the application be installed). After installing, uninstalling or modifying this file, the application must run the 'update-mime-database' command, which is provided by the freedesktop.org shared database[83].

'update-mime-database' is passed the '`mime`' directory containing the '`packages`' subdirectory which was modified as its only argument. It scans all the XML files

```
<?xml version="1.0"?>
<mime-info xmlns='http://www.freedesktop.org/standards/shared-mime-info'>
  <mime-type type="text/x-diff">
    <comment>Differences between files</comment>
    <comment xml:lang="af">verskille tussen lers</comment>
    ...
    <magic priority="50">
      <match type="string" offset="0" value="diff\t"/>
      <match type="string" offset="0" value="***\t"/>
      <match type="string" offset="0" value="Common subdirectories: "/>
    </magic>
    <glob pattern="*.diff"/>
    <glob pattern="*.patch"/>
  </mime-type>
</mime-info>
```

Figure 8.1: An example source file, named 'diff.xml'.

in the '`packages`' subdirectory, combines the information in them, and creates a number of output files.

Any file named '`Override.xml`' takes precedence over all other files in the same '`packages`' directory. This can be used by tools which let the user edit the database to ensure that the user's changes take effect.

Although our system resolves conflicts when conflicting information is provided by different packages, this should be rare because information relating to common types (such as MS Word documents) will be provided in the X Desktop Group's package[83], which all applications following this specification must depend on. Therefore, applications will normally only be providing information about their own types.

8.3.3 The source XML files

Each application provides only a single XML source file, which is installed in the '`packages`' directory as described above. This file is an XML file whose document element is named `<mime-info>` and whose namespace URI is `http://www.freedesktop.org/standards/shared-mime-info`. All elements described in this specification must have this namespace too.

The document element contains zero or more `<mime-type>` child nodes, in any order, each describing a single MIME type. Each element has a '`type`' attribute giving the MIME type that it describes. Figure 8.1 shows a sample source file.

Each `<mime-type>` node may contain any combination of the following elements, and in any order:

- <glob> elements have a 'pattern' attribute. Any file whose name matches this pattern will be given this MIME type (subject to conflicting rules in other files, of course).
- <magic> elements contain a list of <match> elements, any of which may match, and an optional 'priority' attribute for all of the contained rules. Low numbers should be used for more generic types (such as 'gzip compressed data') and higher values for specific subtypes (such as a word processor format that happens to use gzip to compress the file). The default priority value is 50, and the maximum is 100.

  Each <match> element has a number of attributes:

  type This gives the format of the 'value' attribute. It is can be `string`, `host16`, `host32`, `big16`, `big32`, `little16`, `little32` or `byte`. 'big' indicates that the number is stored in the file in big-endian order, 'little' indicates little-endian order, and 'host' indicates that the order is the host's natural order[2].

  offset The byte offset(s) in the file to check. This may be a single number or a range in the form 'start:end', indicating that all offsets in the range should be checked. The range is inclusive.

  value The value to compare the file contents with, in the format indicated by the 'type' attribute.

  mask (optional) The number to bitwise-and the value in the file with before comparing it to 'value'. Masks for numerical types can be any number, while masks for strings must be in base 16, and start with 0x.

  Each element corresponds to one line of *file*'s 'magic.mime' file. They can be nested in the same way to provide the equivalent of continuation lines.
- <comment> elements give a human-readable textual description of the MIME type. There may be many of these elements with different 'xml:lang' attributes to provide the text in multiple languages.
- <root-XML> elements have 'namespaceURI' and 'localName' attributes. If a file is identified as being an XML file, these rules allow a more specific MIME type to be chosen based on the namespace and local name of the document element. If 'localName' is present but empty then the document element may have any name, but the namespace must still match. If 'namespaceURI' is present but empty, the element is not namespaced. It is not legal for both attributes to be empty.

[2]There was some disagreement about whether 'host' matches were useful, since a file should get the same type whatever the platform of the checker. However, all previous systems supported this behaviour (mainly used for shared libraries and other non-portable objects), and it was decided to keep it.

Applications may also define their own elements, provided they are namespaced to prevent collisions. Unknown elements are copied directly to the output XML files in the same way as `<comment>` elements. A typical use for this would be to indicate the default handler application for a particular desktop ("Galeon is the GNOME default text/html browser"). Note that this doesn't indicate the user's *preferred* application, only the (fixed) default.

8.3.4 Generated files

The following files are created by the 'update-mime-database' command, within the '`mime`' directory given as its first argument:

- '`globs`' (contains a mapping from names to MIME types)
- '`magic`' (contains a mapping from file contents to MIME types)
- '`XMLnamespaces`' (contains a mapping from XML elements to MIME types)
- '`MEDIA/SUBTYPE.xml`' (one file for each MIME type, giving details about the type)

The formats of these generated files are explained in the following sections.

Having a program to process the input files serves several purposes. First, it allows applications to get the data they need quickly, without parsing all the source XML files (the base package alone is over 700K). Secondly, it allows the database to be used for other purposes (such as creating the legacy '`/etc/mime.types`' file if desired). Thirdly, it allows validation to be performed on the input data, and removes the need for other applications to carefully check for, and report, errors in the input.

8.3.5 The MEDIA/SUBTYPE.xml files

These files have a `<mime-type>` element as their document element node. The format is as described above. They are created by merging all the `<mime-type>` elements from the source files and creating one output file per MIME type. Each file may contain information from multiple source files. The `<magic>`, `<glob>` and `<root-XML>` elements will have been removed, since they are used to *determine* the type, rather than being information *about* the type.

The example source file given above would, on its own, create an output file called '`<MIME>/text/x-diff.xml`' containing the following:

```
<?xml version="1.0" encoding="utf-8"?>
<mime-type type="text/x-diff"
           xmlns="http://www.freedesktop.org/standards/shared-mime-info">
  <!--Created automatically by update-mime-database. DO NOT EDIT!-->
  <comment>Differences between files</comment>
```

```
  <comment lang="af">verskille tussen lers</comment>
  ...
</mime-type>
```

8.3.6 The glob files

This is a simple list of lines containing a MIME type and pattern, separated by a colon. For example:

```
# This file was automatically generated by the
# update-mime-database command. DO NOT EDIT!
...
text/x-diff:*.diff
text/x-diff:*.patch
...
```

Applications must first try a case-sensitive match, then try again with the filename converted to lower-case if that fails. This is so that a file named '`main.C`' will be seen as a C++ file (overriding the '*.c' pattern for plain C files), while '`IMAGE.GIF`' will still use the '*.gif' pattern.

If several patterns match then the longest pattern should be used. In particular, files with multiple extensions such as '`Data.tar.gz`' must match the longest sequence of extensions (matching '*.tar.gz' in preference to '*.gz' in this case). Literal patterns such as 'Makefile' must be matched before all others. It is acceptable to match patterns of the form '*.text' before other wildcarded patterns, allowing these to be special-cased using a hash table. In practice, this covers the vast majority of name matching rules.

There may be several rules mapping to the same type. If the same pattern is defined twice, then the rules must be ordered by the directory the rule came from, as described in the Base Directory Specification[7].

8.3.7 The magic files

The magic data is stored in a binary format for ease of parsing. The old magic database had complex escaping rules which are now handled by the 'update-mime-database' command. Updating Geoff Youngs' freemime library[3] from the old text based format to the new binary format required a patch which added 473 lines, and removed 2016. The total size of the library after this was 1026 lines. Thus, the binary format makes it much easier for people to implement the specification, and removes a likely source of bugs.

The file starts with the magic string '`MIME-Magic\0\n`'. There is no version number in the file. Incompatible changes will be handled by creating both the

[3] The library is available from the MIME Specification page[83].

current 'magic' file and a newer 'magic2' in the new format. Where possible, only compatible changes will be made.

The rest of the file is made up of a sequence of small sections. Each section is introduced by giving the priority and type in brackets, followed by a newline character. Higher priority entries come first. A section heading looks like this:

```
[50:text/x-diff]\n
```

Each line in the section takes the form:

```
[ indent ] ">" start-offset "=" value
[ "&" mask ] [ "~" word-size ] [ "+" range-length ] "\n"
```

Part	Example	Meaning
indent	`1`	The nesting depth of the rule, corresponding to the number of '>' characters in the traditional 'file' format.
">" start-offset	`>4`	The offset into the file to look for a match.
"=" value	`=\0x0\0x2 \0x55\0x40`	Two bytes giving the (big-endian) length of the value, followed by the value itself.
"&" mask	`&\0xff\0xf0`	The mask, which (if present) is exactly the same length as the value.
"~" word-size	`~2`	On little-endian machines, the size of each group to byte-swap.
"+" range-length	`+8`	The length of the region in the file to check.

Note that the value, value length and mask are all binary, whereas everything else is textual. Each part begins with a single character to identify it, except for the indent level.

The word size is used for byte-swapping. Little-endian systems should reverse the order of groups of bytes in the value and mask if this is greater than one. This only affects 'host' matches ('big32' entries still have a word size of 1, for example, because no swapping is necessary, whereas 'host32' has a word size of 4).

The indent, range-length, word-size and mask components are optional. If missing, indent defaults to 0, range-length to 1, the word-size to 1, and the mask to all 'one' bits.

Indent corresponds to the nesting depth of the rule. Top-level rules have an indent of zero. The parent of an entry is the preceding entry with an indent one less than the entry.

If an unknown character is found where a newline is expected then the whole line should be ignored (there will be no binary data after the new character, so the next line starts after the next '\n' character). This is for future extensions.

The 'text/x-diff' above example would (on its own) create this magic file:

```
00000000  4d 49 4d 45 2d 4d 61 67  69 63 00 0a 5b 35 30 3a  |MIME-Magic..[50:|
00000010  74 65 78 74 2f 78 2d 64  69 66 66 5d 0a 3e 30 3d  |text/x-diff].>0=|
00000020  00 05 64 69 66 66 09 0a  3e 30 3d 00 04 2a 2a 2a  |..diff..>0=..***|
00000030  09 0a 3e 30 3d 00 17 43  6f 6d 6d 6f 6e 20 73 75  |..>0=..Common su|
00000040  62 64 69 72 65 63 74 6f  72 69 65 73 3a 20 0a     |bdirectories: .|
```

8.3.8 The XMLnamespaces files

Each '`XMLnamespaces`' file is a list of lines in the form:

`namespaceURI " " localName " " MIME-Type "\n"`

For example:

`http://www.w3.org/1999/xhtml html application/xhtml+xml`

The lines are sorted (using strcmp) and there are no lines with the same namespaceURI and localName in one file. This means that if '`localName`' was empty then there will be two spaces following the namespace URI.

8.3.9 Storing the MIME type using Extended Attributes

An implementation may also get a file's MIME type from the '`user.mime_type`' extended attribute. The attr(5) man page documents this name. The type given here should normally be used in preference to any guessed type, since the user is able to set it explicitly. Applications may choose to set the type when saving files. Since many applications and filesystems do not support extended attributes, implementations must not rely on this method being available.

8.3.10 Recommended checking order

Because different applications have different requirements, they may choose to use the various methods provided by this specification in any order. However, the recommended order to perform the checks is:

- If a MIME type is provided explicitly (eg, by a ContentType HTTP header, a MIME email attachment header, an extended attribute or by some other means) then that should be used instead of guessing.
- If no explicit type is present, the glob rules should be applied to the name to get the type.
- If no glob rules match, the magic rules should be tried next.
- If nothing matches, the default type of 'application/octet-stream' should be used for binary data, or 'text/plain' for textual data (checking the start of the file for ASCII control characters is a good way to guess whether a file is binary

or text). Note that files with high-bit-set characters should still be treated as text since these can appear in UTF-8 text, unlike control characters.

There are several reasons for checking the glob patterns before the magic. First, some applications don't check the magic at all, and this ordering makes it more likely that both will get the same type. Secondly, users can easily understand why calling their text file '`README.mp3`' makes the system think it's an MP3, whereas they have trouble understanding why their computer thinks '`README.txt`' is a PostScript file (because it happens to start with the two characters '`%!`'). Finally, if the system guesses wrongly, the user can often rename the file to fix the problem.

If desired, magic rules with a high priority can be tried before the glob matching. This should be used for types with a very distinctive signature. It is suggested that such rules should use a priority of 80 or higher.

8.3.11 Security implications

The system described in this document is intended to allow different programs to see the same file as having the same type. This is to help interoperability. The type determined in this way is only a guess, and an application must not trust a file based simply on its MIME type. For example, a downloader should not pass a file directly to a launcher application without confirmation simply because it has a 'harmless' type such as 'text/plain'.

Applications must not rely on two applications getting the same type for the same file, even if they both use this system. The specification allows some leeway in implementation, and in any case the programs may be following different versions of the specification.

8.3.12 User modification

The MIME database is not intended to store user preferences. If users wish to make corrections or to provide MIME entries for software that doesn't provide these itself, they should do so by means of the '`Override.xml`' file mentioned in section 8.3.2. We have also created the *MIME-Editor*[84] application (shown in figure 8.2) which provides a graphical interface for these editing operations.

Information such as "text/html files need to be opened with Mozilla" should not go in this database, but in a configuration system, such as *gconf*[105][4].

[4] Getting all the desktop environments to standardise on a common configuration system is left as an exercise for the reader.

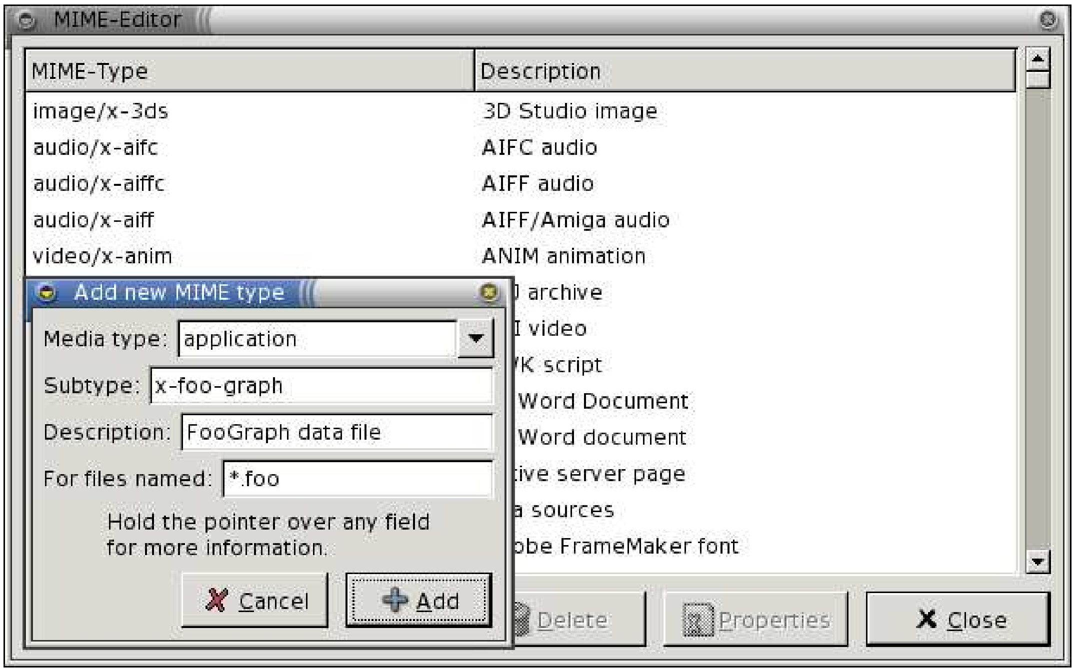


Figure 8.2: The MIME-Editor application.

8.4 Conclusions

In this chapter a new system for storing MIME information has been described. We have proposed our system to the community, and it has been reviewed and endorsed by the X Desktop Group and accepted as the new standard by GNOME, KDE and ROX. By using XML we have been able to check the source files thoroughly, allowing us to detect and report invalid entries. By using XML namespaces, we allow other specifications to build on this one without introducing name collisions or incompatibility.

The existing databases have been merged into a single package[83] by combining the GNOME and KDE databases, resolving conflicts using the type names used by Apache and the *IETF* (*The Internet Engineering Task Force*) as the decider where there were conflicts. A number of errors in the existing databases were discovered and fixed during this process.

The shared database and our 'update-mime-database' program have been packaged by the Debian, Mandrake and Gentoo Linux distributions at the time of writing, and are also available in the official 'ports' archives for both FreeBSD and OpenBSD. RedHat have contributed code to determine the type of a file using this specification for the GTK library's new GtkFileChooser widget (used when loading and saving files in GTK applications).

Chapter 9

Summary

This thesis has presented the case for *tree-sheets*; visual end-user tools for processing XML. We believe that XML is, and will continue to be, sufficiently pervasive that such tools will become as important for handling many types of document as the spreadsheet has proven to be for the processing of numerical data.

In chapter 2, we told of the history of XML and its predecessors, presented its syntax, including namespaces and DTDs, and examined some of the key operations that can be performed on XML documents: validation, querying and transformation.

We presented some current and future uses for XML in chapter 3, and in particular its use in various Internet-related technologies, which take advantage of XML's ability to integrate heterogeneous systems easily. Web page authors can, using XML and stylesheets, give documents clear semantic meaning, rather than focusing on presentational information as the use of traditional HTML forces them to do. We have seen how web pages thus designed provide information that is useful to both human readers and to automatic processing systems.

This gives XML a vital rôle in the creation of the Semantic Web ("*an extension of the current web in which information is given well-defined meaning*") in addition to its use as the most common serialisation format for RDF, which formally encodes facts as relations and uses ontology languages to describe and connect different vocabularies used around the world.

A set of sample use cases for a tree-sheet was presented in chapter 4, and traditional text-based tools were found to be inadequate for these tasks. Dome was introduced, and its operation explained, by showing how these tasks could be performed using it. Dome programs are created by users performing normal direct-manipulation operations on a source document. These operations are designed to make the operations replayable on future, similar documents.

In chapter 5, Dome was used to harvest information from web pages, producing output in RDF. Web pages are typically marked up using the older HTML (or its direct XML equivalent, XHTML), and so require a powerful processing system such as Dome to extract information reliably. We also saw how Dome is able to communicate with remote servers, using both the Hyper-Text Transport Protocol (HTTP) and using XML SOAP messages. Dome is able to present this as a familiar direct-manipulation style of operation, by its technique of replacing a *request* with its *result* in the tree.

The extraction process is simple enough that users can quickly learn to use it, and powerful enough to solve complicated problems, such as merging data from several pages, getting information through online HTML forms, and coping with differing page structures.

The initial experiments revealed some speed problems with both interactive and batch use, but these have been addressed through the use of much faster library code, and other optimisations to Dome itself.

Dome was compared with the Web Consortium's XSLT language in chapter 6. We saw that, for many tasks, either system can be used, but that while an XSLT solution to a problem involves describing how to create a new output document by taking values from the input, a typical Dome solution has the user modifying the input document in-place, turning it into the final output. This style of interaction is more familiar to most users, since it mirrors the behaviour of standard editor applications. However, Dome was also able to support XSLT's before-and-after style of transformation, which we demonstrated by building an XSLT-to-Dome conversion tool.

In chapter 7, we provided an introduction to the field of visual languages, and in particular to the spreadsheet and some of the many attempts to extend it. We have looked at the Forms/3 language, which generalises the spreadsheet from a grid layout to any collection of cells, and at work which replaces the spreadsheet's textual formulae with visual Prograph expressions. We have also seen that Pygmalion shares Dome's approach of editing an initial state into a final state and recording the operations for later application to other input data.

We presented our two earlier efforts to create a visual language, and showed how these experiences also influenced the development of Dome, in particular by highlighting the need for visual data-structures. The earliest of these, VVSh, was purely a scripting tool, used to 'glue' different programs together. The second one used threads, linking smaller components together but still requiring textual programing for the individual pieces. Dome is a full programming language, capable of performing a wide variety of tasks without recourse to another language.

We then described uses for Dome which exploit its power as a general purpose programming language, rather than as a transformation language for external XML data, which had been covered in previous chapters. We saw that, despite having no intelligent learning algorithms, programs such as 'quicksort' can be implemented quite naturally by demonstration in Dome.

We have also shown that a Dome program can provide a fully visual custom interface, using a canvas widget, and that the internal state information can be designed to follow closely the graphical output, making the program easier to understand.

In chapter 8, a new system for storing MIME information was described. We have proposed our system to the community, and it has been reviewed and endorsed by the X Desktop Group and accepted as the new standard by GNOME, KDE and ROX. By using XML we have been able to thoroughly check the source files, allowing us to detect and report invalid entries. By using XML namespaces, we allow other specifications to build on this one without introducing name collisions or incompatibility.

9.1 Contributions

We have created a new tree-sheet, Dome. Unlike other XML manipulation tools, Dome is both a flexible and programmable editor, and has an easy-to-use graphical interface using a direct manipulation style. Dome's use of a flat list of namespaces to resolve the many well-known scoping issues is also novel, as far as we are aware.

Dome is unique in its use of a tree structure to nest operations; processing a subpage on a site is presented as editing the expansion of one of the links in the site index, keeping the whole operation in context. Dome's editing model is simple enough to be quickly grasped by users, but, thanks to the flexibility of the tree structure, powerful enough to handle demanding tasks.

A number of separate and incompatible MIME databases have been combined, using XML, to create a flexible and extensible shared database. This work is in the process of being adopted by many parties, and is also being used as the basis for other integration projects at freedesktop.org, including the *Icon theme system*[77] (which requires a common set of MIME type names so that the same icon is used for each type) and the planned *MIME actions system*[1] (which extends the database to include information about which applications can handle which MIME types). A unified MIME type database is also a requirement for reliable drag-and-drop and cut-and-paste operations between programs.

[1]http://www.freedesktop.org/Standards/mime-actions-spec

Appendix A

Code fragments

A.1 XML Pull example

This is the example from section 2.2.6, with the missing `filter_elements` function included. The function is generic enough that it should probably form part of the core PullXML API. It filters the stream of XMLPull events to only include elements with the given name. Textual data and elements with other names are skipped.

```
def filter_elements(parser, name):
    for token, node in parser:
        if token == 'START_ELEMENT':
            if not name or node.localName == name: yield node
            else: parser.expandNode(node)
        elif token == 'END_ELEMENT': return


n = 1
filter_elements(parser, None).next()
for chapter in filter_elements(parser, 'chapter'):
    for title in filter_elements(parser, 'title'):
        print "Chapter %d: %s" % (n, parser.getEvent()[1].data)
        n += 1
        parser.expandNode(title)
```

A.2 SAX example

This is the SAX parser equivalent to the XML-Pull parser given in section 2.2.6. The event-based nature of the SAX system requires us to store the current state in an object across calls, rather than using the natural control structures of the language, resulting in code that is longer and much harder to read.

```
class handler(sax.ContentHandler):
    def __init__(self):
        self.depth = 0
        self.n = 1

    def startElement(self, name, attrs):
        self.depth += 1
        if self.depth == 2:
            self.in_chapter = (name == 'chapter')
        if self.depth == 3 and self.in_chapter:
            self.in_title = (name == 'title')
            self.title = ''
        else:
            self.in_title = False

    def endElement(self, name):
        self.depth -= 1
        if self.in_title and name == 'title':
            self.in_title = False
            print "Chapter %d: %s" % (self.n, self.title)
            self.n += 1
        elif self.in_chapter and name == 'chapter':
            self.in_chapter = False

    def characters(self, data):
        if self.in_title:
            self.title += data
```

Bibliography

[1] Adler, S., Berglund, A., Caruso, J., Deach, S., Graham, T., Grosso, P., Gutentag, E., Milowski, A., Parnell, S., Richman, J. and Zilles, S. *Extensible Stylesheet Language (XSL) Version 1.0*, Oct 2001.

[2] Altova *xmlspy 5 Online Manual.*
`http://www.xmlspy.com/manual`

[3] Apache Software Foundation *The Apache HTTP Server.*
`http://www.apache.org`

[4] Apache Software Foundation *Xerces: XML parsers in Java and C++.*
`http://xml.apache.org/#xerces`

[5] Ausbrooks, R., Buswell, S., Dalmas, S., Devitt, S., Diaz, A., Hunter, R., Smith, B., Soiffer, N., Sutor, R. and Watt, S. *Mathematical Markup Language (MathML) Version 2.0*, Feb 2001.
`http://www.w3.org/TR/MathML2/`

[6] B. Ibach, I. Castle, D. Martin and P. N. Lundblad *OpenJade Project.*
`http://openjade.sourceforge.net/`

[7] Bastian, W. *XDG Base Directory Specification*, Mar 2003. The X Desktop Group.
`http://www.freedesktop.org/standards`

[8] Baumgartner, R., Flesca, S. and Gottlob, G. *Visual Web Information Extraction with Lixto*, The VLDB Journal, pages 119-128, 2001.
`citeseer.nj.nec.com/baumgartner01visual.html`

[9] Baumgartner, R., Flesca, S. and Gottlob, G. *Supervised Wrapper Generation with Lixto*, The VLDB Journal, pages 715-716, 2001.
`citeseer.nj.nec.com/baumgartner01supervised.html`

[10] Berners-Lee, T. *Uniform Resource Identifiers (URI): Generic Syntax*, Aug 1998.
`http://www.ietf.org/rfc/rfc2396.txt`

[11] Berners-Lee, T., Hendler, J. and Lassila, O. *The Semantic Web*, Scientific American, May 2001.

[12] Bernstein, D. J. *Internationalized domain names*, Nov 2001.
`http://cr.yp.to/djbdns/idn.html`

[13] Bonifati, A. and Ceri, S. *Comparative Analysis of Five XML Query Languages*, SIGMOD Record, number 1, pages 68-79, volume 29, 2000.
`citeseer.nj.nec.com/bonifati00comparative.html`

[14] Borenstein, N. and Freed, N. *RFC1521: MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for Specifying and Describing the Format of Internet Message Bodies.*
`http://www.faqs.org/rfcs/rfc1521.html`

[15] Bos, B., Wium Lie, H., Lilley, C. and Jacobs, I. *Cascading Style Sheets, level 2 CSS2 Specification*, May 1998.

[16] Bourret, R., Cowan, J., Macherius, I. and S. St. Laurent *Document Definition Markup Language (DDML) Specification, Version 1.0*, Jan 1999.
`http://www.w3.org/TR/1999/NOTE-ddml-19990119`

[17] Bourret, R. *XML Namespaces FAQ*, Feb 2003.
`http://www.rpbourret.com/xml/NamespacesFAQ.htm`

[18] Bourret, R. *Namespace Myths Exploded*, Mar 2000.
`http://www.xml.com/pub/a/2000/03/08/namespaces/index.html`

[19] Bray, T., Frankston, C. and Malhotra, A. *Document Content Description for XML*, Jul 1998.
`http://www.w3.org/TR/NOTE-dcd`

[20] Bray, T., Hollander, D. and Layman, A. *Namespaces in XML*, Jan 1999.
`http://www.w3.org/TR/REC-xml-names/`

[21] Bray, T. *Why XML Doesn't Suck*, Mar 2003.
`http://tbray.org/ongoing/When/200x/2003/03/24/XMLisOK`

[22] Brickley, D. and Guha, R. V. *RDF Vocabulary Description Language 1.0: RDF Schema (draft)*, Jan 2003.
`http://www.w3.org/TR/rdf-schema/`

[23] Brown, P., Blandford, J. and Taylor, O. *Desktop Entry Specification.*
`http://www.freedesktop.org/standards/desktop-entry-spec.html`

[24] Bry, F. and Schaffert, S. *The XML Query Language Xcerpt: Design Principles, Examples, and Semantics.*
citeseer.nj.nec.com/bry02xml.html

[25] Buneman, P. *Semistructured data*, pages 117–121, 1997.
citeseer.nj.nec.com/buneman97semistructured.html

[26] Burnett, M. *A Guided Tour Of Forms/3*, Apr 2001.
http://web.engr.oregonstate.edu/~burnett/Forms3/Tour/tour.html

[27] Burnett, M., Atwood, J., Djang, R., Gottfried, H., Reichwein, J. and Yang, S. *Forms/3: A First-Order Visual Language to Explore the Boundaries of the Spreadsheet Paradigm*, Journal of Functional Programming, pages 155-206, volume 11(2), March 2001.

[28] Burnett, M. M. *Visual Programming*, Encyclopedia of Electrical and Electronics Engineering, 1999.
citeseer.nj.nec.com/burnett99visual.html

[29] Carlson, P. and Burnett, M. *Integrating Algorithm Animation into a Declarative Visual Programming Language*, 11th Intl. IEEE Symposium on Visual Languages, pages 126-127, 1995. IEEE Computer Society Press, Los Alamitos, California, ISSN 1049-2615.

[30] Chamberlin, D., Fankhauser, P., Marchiori, M. and Robie, J. *XML query use cases*, Dec 2001.

[31] Chamberlin, D. *XQuery: An XML query language*, IBM Systems Journal, volume 41, 4, 2002.
http://www.research.ibm.com/journal/sj/414/chamberlin.pdf

[32] Christensen, E., Curbera, F., Meredith, G. and Weerawarana, S. *Web Services Description Language (WSDL) 1.1*, Mar 2001.
http://www.w3.org/TR/wsdl

[33] Clark, J. and Murata, M. *RELAX NG Tutorial*, December 2001.
http://www.oasis-open.org/committees/relax-ng/tutorial-20011203.html

[34] Clark, J. and Murata, M. *RELAX NG Specification*, December 2001.
http://www.oasis-open.org/committees/relax-ng/spec-20011203.html

[35] Clark, J. *Comparison of SGML and XML.*
http://www.w3.org/TR/NOTE-sgml-xml-971215

[36] Clark, J. *XSL Transformations (XSLT) Version 1.0*, Nov 1999.

[37] Clark, J. *The Expat XML Parser.*
http://expat.sourceforge.net/

[38] Cowan, J. *Extensible Markup Language (XML) 1.1*, Oct 2002.

[39] Cowan, J. *XML Blueberry Requirements*, Sep 2001.
`http://www.w3.org/TR/xml-blueberry-req`

[40] Cox, P. T. and Smedley, T. J. *Using visual programming to extend the power of spreadsheet*, Proceedings of the workshop on Advanced visual interfaces, pages 153–161, 1994. ACM Press, 0-89791-733-2.
`http://doi.acm.org/10.1145/192309.192343`

[41] Cypher, A. *Watch What I Do: Programming by Demonstration*, 1993. The MIT Press.

[42] Davidson, A., Fuchs, M., Hedin, M., Jain, M., Koistinen, J., Lloyd, C., Maloney, M. and Schwarzho, K. *Schema for Object-Oriented XML 2.0*, Jul 1999.
`http://www.w3.org/TR/NOTE-SOX/`

[43] Debian Project *The Debian Linux distribution.*
`http://www.debian.org`

[44] Denker, G., Hobbs, J. R., Martin, D., Narayanan, S. and Waldinger, R. J. *Accessing Information and Services on the DAML-Enabled Web*, SemWeb, 2001.
`citeseer.nj.nec.com/denker01accessing.html`

[45] Dodds, L. *Schematron: validating XML using XSLT*, Apr 2001.
`http://www.ldodds.com/papers/schematron_xsltuk.html`

[46] Drake, F. L. *The XML Bookmark Exchange Language, Release 1.0*, October 1998.
`http://pyxml.sourceforge.net/topics/xbel/docs/html/xbel.html`

[47] Dublin Core Metadata Initiative *Dublin Core Metadata Element Set, Version 1.1: Reference Description*, Jul 1999.
`http://dublincore.org/`

[48] Fensel, D., Horrocks, I., van Harmelen, F., Decker, S., Erdmann, M. and Klein, M. C. A. *OIL in a Nutshell*, Knowledge Acquisition, Modeling and Management, pages 1-16, 2000.
`citeseer.nj.nec.com/horrocks00oil.html`

[49] Ford, P. *August 2009: How Google beat Amazon and Ebay to the Semantic Web*, Jul 2002.
`http://www.ftrain.com/google_takes_all.html`

[50] Fourthought Inc. *Open source XML processing tools.*
http://4Suite.org/

[51] Frankston, C. and Thompson, H. S. *XML-Data reduced (Draft)*, Jul 1998.
http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm

[52] Glaser, H. and Smedley, T. *PSH — The Next Generation of Command Line Interfaces*, 11th Intl. IEEE Symposium on Visual Languages, pages 29-36, 1995. IEEE Computer Society Press, Los Alamitos, California, ISSN 1049-2615.

[53] Gnome project *The GNOME desktop.*
http://www.gnome.org

[54] Gnome project *The XML C library for Gnome.*
http://xmlsoft.org/

[55] Goldman, R., Chawathe, S., Cresp, A. and McHugh, J. *A Standard Textual Interchange Format for the Object Exchange Model (OEM)*, Oct 1996.
http://www-db.stanford.edu/~mchughj/oemsyntax/oemsyntax.html

[56] Gudgin, M., Hadley, M., Mendelsohn, N., Moreau, J. and Nielsen, H. F. *SOAP Version 1.2 Part 1: Messaging Framework*, Jun 2002.

[57] Hammer, J., Garcia-Molina, H., Cho, J., Crespo, A. and Aranha, R. *Extracting Semistructured Information from the Web*, Proceedings of the Workshop on Management of Semistructured Data, 1997.
citeseer.nj.nec.com/hammer97extracting.html

[58] Harold, E. R. and Venners, B. *What's Wrong with XML APIs*, May 2003.
http://www.artima.com/intv/xmlapis2.html

[59] Heflin, J. and Hendler, J. *Searching the web with SHOE*, In Artificial Intelligence for Web Search; Papers from the AAAI Workshop, pages 35–40, 2000.
citeseer.nj.nec.com/heflin00searching.html

[60] Heflin, J. *Web Ontology Language (OWL) Use Cases and Requirements*, March 2003.
http://www.w3.org/TR/webont-req/

[61] Hollander, D. and Sperberg-McQueen, C. M. *Happy Birthday, XML!*, Feb 2003.
http://www.w3.org/2003/02/xml-at-5.html

[62] Howe, D. *The Free On-line Dictionary of Computing.*
`http://wombat.doc.ic.ac.uk/foldoc/`

[63] Hughes, J. *Why Functional Programming Matters*, Computer Journal, number 2, pages 98–107, volume 32, 1989.
`citeseer.nj.nec.com/hughes84why.html`

[64] IBM *XSLerator (XSL accelerator)*, Mar 2001.
`http://www.alphaworks.ibm.com/tech/xslerator`

[65] IEEE *IEEE Standard for the Scheme Programming Language*, 1991.

[66] International Organization for Standardization *Information processing – Text and office systems – Standard Generalized Markup Language (SGML).*
`http://www.iso.ch/cate/d16387.html`

[67] International Organization for Standardization *Information Technology — Processing Languages — Document Style Semantics and Specification Language (DSSSL)*, 1996.

[68] International SGML Users' Group *A Brief History of the Development of SGML*, Jun 1990.
`http://www.sgmlsource.com/history/sgmlhist.htm`

[69] Iocchi, L. *The Web-OEM approach to Web information extraction*, Journal of Network and Computer Applications, pages 259-269, volume 22, 1999.

[70] Jabber Software Foundation *Jabber User Guide*, 2002.
`http://www.jabber.org/user/userguide/`

[71] James Clark *RELAX NG and W3C XML Schema*, Jun 2002.
`http://www.imc.org/ietf-xml-use/mail-archive/msg00217.html`

[72] James Clark *Jade - James' DSSSL Engine.*
`http://www.jclark.com/jade/`

[73] KDE team *The KDE desktop.*
`http://www.kde.org`

[74] Klarlund, N., Moller, A. and Schwatzbach, M. I. *"DSD: A Schema Language for XML"*, ACM SIGSOFT Workshop on Formal Methods in Software Practice, 2000. Portland, OR.
`citeseer.nj.nec.com/klarlund00dsd.html`

[75] Koivunen, M., Swick, R., Handschuh, S., Dieng-Kuntz, R. and Staab, S. *Metadata Based Annotation Infrastructure offers Flexibility and*

Extensibility for Collaborative Applications and Beyond, Workshop 4, Knowledge Markup and Semantic Annotation, K-CAP 2001, pages 1–4, Oct 2001.

[76] Kushmerick, N. *Wrapper induction: Efficiency and expressiveness*, Artificial Intelligence, pages 15-68, volume 118, 2000.

[77] Larsson, A. *Icon Theme Specification 0.7*, Sep 2003.
`http://www.freedesktop.org/Standards/icon-theme-spec`

[78] Lassila, O. and Swick, R. R. *Resource Description Framework (RDF)*, Feb 1999.
`http://www.w3.org/RDF/`

[79] Layman, A., Jung, E. and others *XML-Data*, Jan 1998.
`http://www.w3.org/TR/1998/NOTE-XML-data/`

[80] Lee, D. and Chu, W. W. *Comparative analysis of six XML schema languages*, SIGMOD Record (ACM Special Interest Group on Management of Data), number 3, pages 76–87, volume 29, 2000.
`citeseer.nj.nec.com/lee00comparative.html`

[81] Leonard, T., Glaser, H., Handschuh, S., Dieng-Kuntz, R. and Staab, S. *Large scale acquisition and maintenance from the web without source access*, Workshop 4, Knowledge Markup and Semantic Annotation, K-CAP 2001, pages 97-101, Oct 2001. Southampton University.

[82] Leonard, T. *The ROX desktop.*
`http://rox.sf.net`

[83] Leonard, T. *Shared MIME-info Database*, Mar 2003. The X Desktop Group.
`http://www.freedesktop.org/standards/shared-mime-info-spec`

[84] Leonard, T. *MIME-Editor application*, Mar 2003.
`http://rox.sf.net/mime_editor.html`

[85] Leonard, T. *The Choices System*, Jun 2000.
`http://rox.sourceforge.net/choices.html`

[86] Liu, L., Pu, C. and Han, W. *XWRAP: An XML-enabled Wrapper Construction System for Web Information Sources*, ICDE, 2000.

[87] Manola, F. and Miller, E. *RDF Primer*, Jan 2003.
`http://www.w3.org/TR/rdf-primer/`

[88] Mary Fernandez, Simeon, J. and Philip Wadler *XML Query Languages: Experiences and Examplars*, Communication to the XML Query W3C Working Group, Sep 1999.

[89] Mattis, P., Kimball, S., MacDonald, J. and Taylor, O. *The GIMP Toolkit.*
`http://www.gtk.org/`

[90] McCarthy, J. *Recursive Functions of Symbolic Expressions Their Computation by Machine, Part I*, Communications of the ACM, pages 184–195, volume 3, 1960. ACM Press, 0001-0782.
`http://doi.acm.org/10.1145/367177.367199`

[91] Megginson, D. *SAX, the Simple API for XML.*
`http://www.saxproject.org/`

[92] Michail, A. *Teaching Binary Tree Algorithms through Visual Programming*, Proceedings of the IEEE Symposium on Visual Languages, pages 38–45, 1996. IEEE Computer Society PressWashington, 0-8186-7508-X.
`citeseer.nj.nec.com/article/michail96teaching.html`

[93] Mihaila, G. A. *WebSQL: Querying the WWW.*
`http://www.cs.toronto.edu/~websql`

[94] Mockapetris, P. *RFC1034: Domain Names — Concepts and Facilities.*
`http://www.faqs.org/rfcs/rfc1034.html`

[95] Morphon Technologies *Morphon XML Editor Manual 3.1*, Apr 2003.
`http://www.morphon.com/manual.version3/`

[96] Muellner, L. and Walsh, N. *DocBook: The Definitive Guide*, Oct 1999. O'Reilly & Associates, 1-56592-580-7.
`http://docbook.org/`

[97] Muench, S. and Scardina, M. *XSLT Requirements Version 2.0*, Feb 2001.
`http://www.w3.org/TR/xslt20req`

[98] Mller, A. and Schwartzbach, M. I. *Problems with XML Schema*, 2002.
`http://www.brics.dk/~amoeller/XML/schemas/xmlschema-problems.html`

[99] Netscape Communications Corp. *Core JavaScript Guide 1.5*, Sep 2000.
`http://devedge.netscape.com/library/manuals/2000/javascript/1.5/guide/`

[100] Network Working Group *Internationalized Resource Identifiers (IRI) (draft-duerst-iri-03)*, Mar 2003.
`http://www.w3.org/International/iri-edit/`

[101] Newman, C. and Melnikov, A. *ACAP Media Type Dataset Class (draft)*, Dec 2002. Network Working Group.
`ftp://ftp.ietf.org/internet-drafts/draft-ietf-acap-mediatype-01.txt`

[102] Object Management Group *CommonObject Request Broker Architecture: Core Specification 3.0*, Nov 2002.
`http://www.omg.org/cgi-bin/doc?formal/02-11-03`

[103] Ogbuji, U. *Proper XML Output in Python*, XML.com Archives, Nov 2002.
`http://www.xml.com/pub/a/2002/11/13/py-xml.html`

[104] Papakonstantinou, Y., Garcia-Molina, H., Widom, J., Yu, P. S. and A. L. P. Chen *Object Exchange across Heterogeneous Information Sources*, 11th Conference on Data Engineering, pages 251–260, 1995. IEEE Computer SocietyTaipei, Taiwan.
`citeseer.nj.nec.com/article/papakonstantinou95object.html`

[105] Pennington, H. *GConf: Manageable User Preferences*, Proceedings of the Ottawa Linux Symposium, pages 414–424, Jun 2002.
`citeseer.nj.nec.com/article/papakonstantinou95object.html`

[106] Pfeiffer, J. J. *A Rule-Based Visual Language for Small Mobile Robots*, Proceedings IEEE Symposium on Visual Languages, pages 162-163, 1997. IEEE Computer Society Press, Los Alamitos, California.

[107] Pictorius Incorporated *Prograph CPX User's Guide*, 1993. Prograph International.

[108] RSS-DEV Working Group *RDF Site Summary (RSS) 1.0*, May 2001.
`http://web.resource.org/rss/1.0/spec`

[109] Raggett, D. *Assertion Grammars*, May 1999.
`http://www.w3.org/People/Raggett/dtdgen/Docs/`

[110] Raggett, D. *Clean up your Web pages with HTML TIDY.*
`http://www.w3.org/People/Raggett/tidy/`

[111] Rivest, R. *The MD5 Message-Digest Algorithm*, April 1992.
`http://www.faqs.org/rfcs/rfc1321.html`

[112] Rose, M. *RFC3080: The Blocks Extensible Exchange Protocol Core*, Mar 2001.
`http://www.ietf.org/rfc/rfc3080.txt`

[113] Rose, M. *Mapping the BEEP Core onto TCP*, Mar 2001.
`http://www.ietf.org/rfc/rfc3081.txt`

[114] SVG 1.0 Working Group (W3C) *Scalable Vector Graphics (SVG) 1.0 Specification*, Sep 2001.

[115] Sahuguet, A. and Azavant, F. *Looking at the Web through XML glasses*, CoopIs'99, 1999.

[116] Sahuguet, A. and Azavant, F. *Building Intelligent Web Applications Using Lightweight Wrappers*, Data and Knowledge Engineering (to appear), 2000.

[117] Sahuguet, A. and Azavant, F. *Building light-weight wrappers for legacy Web data-sources using W4F*, International Conference on Very Large Databases (VLDB), 1999.

[118] Saint-Andre, P. and Miller, J. *The Extensible Messaging and Presence Protocol (XMPP) (draft)*, Feb 2003.
`http://www.jabber.org/ietf/draft-ietf-xmpp-core-04.html`

[119] Schaffert, S. *Xcerpt Tutorial.*
`http://www.xcerpt.org/tutorial/`

[120] Shadbolt, N. *AKT Selected Papers*, Apr 2003. 0854-327932.
`http://www.aktors.org/publications/selected-papers/`

[121] Shu, N. C. *Visual Programming: Perspectives and Approaches*, IBM Systems Journal, pages 199-221, volume 38, 2/3, 1999.

[122] Slominski, A. *Common API for XML Pull Parsing.*
`http://www.xmlpull.org`

[123] Smith, D., Cypher, A. and Spohrer, J. *Kidsim: programming agents without a programming language*, Communications of the ACM, pages 54-67, volume 37(7), July 1994.

[124] Smitha, M. K., Welty, C. and McGuinness, D. *OWL Web Ontology Language Guide (draft)*, Mar 2003.
`http://www.w3.org/TR/owl-guide/`

[125] Sosnoski, D. *Data binding Part 3: JiBX architecture*, IBM developerWorks, Apr 2003.
`http://www-106.ibm.com/developerworks/library/x-databd3/index.html`

[126] Sperberg-McQueen, C. M. and Burnard, L. *A Gentle Introduction to SGML*, 1994.
`http://www.isgmlug.org/sgmlhelp/g-index.htm`

[127] Sugiura, A. and Koseki, Y. *Simplifying macro definition in programming by demonstration*, Proceedings of the ACM Symposium on User Interface Software and Technology, 1996.

[128] Thompson, K. *Regular expression search algorithm*, Communications of the ACM, volume 11–6, Jun 1968.

[129] UDDI Spec Technical Committee *UDDI Version 3.0*, Jul 2002.
`http://uddi.org/pubs/uddi-v3.00-published-20020719.htm`

[130] Unicode Consortium *Unicode 3.2.0*, Mar 2002.
`http://www.unicode.org/`

[131] Van Rossum, G. *The Python programming language*.
`http://www.python.org/`

[132] W3C HTML working group *XHTML 1.0: The Extensible HyperText Markup Language*, Jan 2000.

[133] World Wide Web Consortium *Platform for Internet Content Selection (PICS)*.
`http://www.w3.org/PICS/`

[134] World Wide Web Consortium *XForms 1.0*.
`http://www.w3.org/TR/xforms/`

[135] World Wide Web Consortium *HyperText Markup Language*.
`http://www.w3.org/MarkUp/`

[136] World Wide Web Consortium *Extensible Markup Language (XML) draft*.
`http://www.w3.org/TR/WD-xml-961114.html`

[137] World Wide Web Consortium *XML Query*.
`http://www.w3.org/XML/Query`

[138] World Wide Web Consortium, Thompson, H. S., Beech, D., Maloney, M. and Mendelsohn, N. *XML Schema Part 1: Structures*, Oct 2001.
`http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/`

[139] World Wide Web Consortium, Cowan, J. and Tobin, R. *XML Information Set*, Oct 2001.
`http://www.w3.org/TR/xml-infoset/`

[140] World Wide Web Consortium *Extensible Markup Language (XML)*.
`http://www.w3.org/XML/`

[141] World Wide Web Consortium *Document Object Model (DOM) Level 2 Core Specification*.
`http://www.w3.org/TR/DOM-Level-2-Core/`

[142] World Wide Web Consortium *XPointer Framework*, Nov 2002.
`http://www.w3.org/TR/xptr-framework/`

[143] World Wide Web Consortium *XML Linking Language (XLink) Version 1.0*, Jun 2001.
`http://www.w3.org/TR/xlink/`

[144] World Wide Web Consortium *XML Path Language (XPath)*.
`http://www.w3c.org/TR/xpath`

[145] World Wide Web Consortium *Metadata Activity Statement*, Aug 2002.
`http://www.w3.org/Metadata/Activity`

[146] X Consortium *X Window System*.
`http://xwin.org`

[147] Yergeau, F. *UTF-8, a transformation format of ISO 10646*, Jan 1998.
`ftp://sunsite.doc.ic.ac.uk/packages/rfc/rfc2279.txt`

[148] Zoulas, C. *The Unix 'file' utility*.
`http://freshmeat.net/projects/file/?topic_id861`

Index