XML in the Visualisation Pipeline

Warwick Irwin **Neville Churcher**

Software Visualisation Group,
Department of Computer Science,
University of Canterbury,
Private Bag 4800, Christchurch, New Zealand.
E-mail: {wal,neville}@cosc.canterbury.ac.nz

Abstract

The extensible markup language (XML) has been applied successfully in a wide range of application domains and is beginning to find applications in visualisation. We explore the possibility that, rather than simply being used as an exchange format, XML might become a fundamental medium in an extended visualisation pipeline. We present examples from our software visualisation research to illustrate that increased rigour, as well as flexibility, can be delivered by integrating XML into the visualisation pipeline.

Keywords: Visualisation, XML, VRML

1 Introduction

The visualisation pipeline is a convenient metaphor for the representation, transformation and presentation of data. It has served the visualisation community well for many years and provides a foundation for further developments. Many current visualisation toolkits and packages are built on this fundamental model (Schroeder, Martin & Lorensen 1998, for example).

In recent years a number of significant shifts have occurred, across a wide range of software development fields, which are relevant to the evolution of the pipeline model of visualisation. Of particular importance is an increased expectation that end users, the consumers of visualisations, will play a more central rôle.

Users in a wide range of fields increasingly expect seamless integration of resources located anywhere on the Internet. Further, users expect flexibility and the ability to customise and perform exploratory analysis.

Similar problems have been encountered in fields other than visualisation. The problems arising from the need to exchange data and metadata between incompatible Computer Aided Software Engineering (CASE) tools led to the development of standards such as the CASE Data Interchange Format (CDIF). Similar problems arise in Geographic Information Systems (GIS).

XML (XML 2001, Martin, Birkbeck, Kay, Loesgen, Pinnock, Livingstone, P.Stark, Williams, Anderson, Mohr, Baliles, Peat & Ozu 2000) addresses these and other outstanding issues. XML-based solutions to the CASE tool data exchange (XMI) and GIS exchange (ISO technical committee TC211) problems are under active development.

However, there is a danger that the full benefits of XML will not be realised if we do not learn from the failures of previous attempts to establish standard formats. Standards such as CDIF ultimately failed for two reasons. Firstly, they were not extensible—CDIF's days were numbered when object oriented techniques reached the mainstream. Secondly, they were primarily bulk export formats—producing large files of "dead" *data* as the end result of all the processing activity which produced the corresponding *information*.

One of the major difficulties we have encountered in our software visualisation research (Churcher, Keown & Irwin 1999, Hartley, Churcher & Albertson 2000) is the determination of the degree of rigour associated with a particular visualisation. In particular, the parsing of real languages involves challenges such as handling ambiguities in their grammars and determining (without guessing) the types of identifiers. In practice, there is considerable variation in the level of rigour of data reported in the literature and it is tempting to simply ignore the awkward issues. However, we argue that these are in fact often the most vital to the complete understanding of the systems in question—a major goal of any visualisation.

Our approach is motivated by concerns such as those expressed by Griswold, describing his work on program slicing, who said (sic) *... we found that if the precision of the various parts were not balanced, then the benefits of the more precise (and costlier) components was not realized* (Griswold 2001).

If a standard format is used throughout the process then it becomes possible for tools to communicate at a semantic level and to exchange and manipulate data in units directly related to the tasks at hand.

Further benefits accrue from the exposure of the internals of the pipeline. In particular, it becomes possible to visualise readily the intermediate products at various stages of the pipeline.

The remainder of the paper is structured as follows. In the next section we give a brief overview of the relevant aspects of XML and related technologies and outline how they fit into the software visualisation pipeline. Section 3 describes our expanded view of the pipeline model. Section 4 discusses relevant aspects of parsing and describes yakyacc, a tool we have built as part of our work on the use of XML in software visualisation. Transformations are discussed in Section 5 and some examples of results from our work are presented in Section 6. Our conclusions and suggestions for further work appear in Section 7.

2 XML in the Software Visualisation Pipeline

XML is a deceptively simple language. Superficially, it resembles HTML, having the familiar `<element att='value'>content</element>` syntax and a text-based format. Advantages include extensibility, allowing users to design their own domain-

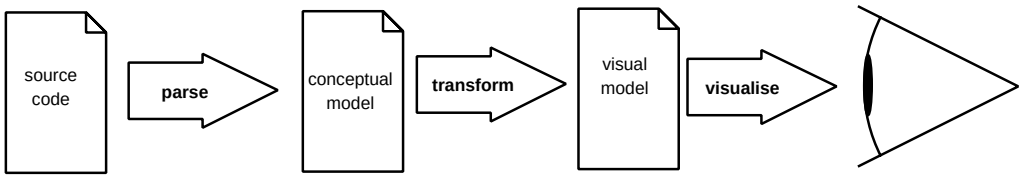

Figure 1: Simplified pipeline

specific vocabularies by defining additional elements and attributes. Many such vocabularies are emerging. In addition, the rules governing the relationships between elements and other document components may be specified by means of a grammar.

A grammar may be defined in one of two formats: as a document type definition (DTD), whose syntax derives from SGML, or as a schema definition, whose syntax is that of XML itself. The language syntax makes it possible to determine whether or not a document is well-formed (legal) and—if a grammar is available—correct (meaningful).

This allows powerful, yet general, parsing and translation tools to be developed. A wide range of software, both commercial and open-source, is available. There are two main approaches. A document object model (DOM) parser constructs the entire document structure in memory. This simplifies access but is potentially prohibitively costly in resources. Event driven (SAX) parsers trigger actions when interesting events, such as occurrences of `</element>`, occur—the entire DOM tree need never be constructed explicitly and the parsing may be embedded in a stream of communication between processes. A further option is provided by XSLT stylesheet transformations (described further in Section 5) which allow complex transformations (XML → XML; XML → LaTeX; XML → VRML; ...) to be defined without the need to construct a parser explicitly.

Our software visualisation research involves the capture of data about the detailed properties of software. Ultimately, this data is visualised in virtual worlds *via* the virtual reality modelling language (VRML) (Carey & Bell 1997, Churcher et al. 1999).

We wish to express this process—from beginning to end—as a sequence of transformations which convert data, captured by a variety of tools, into well-defined internal representations and ultimately into forms, such as VRML, ready for rendering.

A number of intermediate stages in this sequence of transformations may be identified and specific artifacts are associated with each:

Grammars for the programming languages being studied (e.g. Java, C++)

Automata describing the basic structure of corresponding parsers

Generated parsers capable of fundamental parsing activities

Instrumented parsers capable of recording data about software metrics and other properties

Parse trees resulting from running instrumented parsers on source code

Transformed data resulting from processing of parse tree data into generic form

Visualisation models suitable for individual visualisation techniques such as treemaps (Johnson & Shneiderman 1991)

Visualisation data (typically VRML in our work) ready for use.

XML and its related technologies are suitable for use at each stage. For example, we represent grammars in XML and use XSLT to transform parse tree data to visualisation models. We are thus able to use a consistent set of representations and to perform a wide range of transformations without loss of information.

In addition to these considerable benefits, we are also able to visualise the intermediate stages in our pipeline—such as the structure of the state machine—with little extra effort.

A further significant advantage is the ability to expose the precise details of grammars and other artifacts. This allows calibration of our tools and simplifies the comparison of experimental work by different researchers.

3 Expanded Visualisation Pipeline

In order to experiment with software visualisations, we required a toolset capable of examining software source and exposing its structure using visual models. Flexibility was an important goal: the toolset would ideally handle source code written in a number of programming languages (particularly Java, C and C++) and would be capable of producing diverse visualisation models.

We chose a pipeline architecture, in which the source code undergoes a sequence of transformations to produce a visualisation model. Figure 1 shows a simplified pipeline. The first phase parses the source code and emits a syntax tree as an XML file. The second phase transforms the parse tree into a visual model (usually another XML file) suitable for input into the third phase, a tool that renders the visualisation.

The advantage of a pipeline architecture is the flexibility it affords. Each tool along the pipeline can be substituted for another, with minimal impact on downstream transformations. Thus, an appropriate parser can be chosen for the source code and the output can be adapted to suit an arbitrary visualisation tool.

The central step in the pipeline, transform, decouples the parsing tools from the visualisation tools. The transformation maps a conceptual model to a visualisation model. This mapping may translate from one file format to another. For example, it may translate an XML representation of a syntax tree into an XML graph description. The mapping may also select a subset of data relevant to a particular visualisation, for example, by extracting a method call graph from a parse tree.

The task of transforming XML files is supported by a rich set of commonly available utilities including XML parsers and writers, and in particular, XSLT (eXtensible Stylesheet Language: Transformations). XSLT allows transformations of XML files into other XML files (or arbitrary file formats), without the need for conventional programming. As depicted in Figure 2, the transformation is accomplished by an XSLT processor, which applies an XSLT stylesheet to an XML source file to produce a result file. An XSLT

stylesheet is a high-level declarative specification of the behaviour of the transformation.

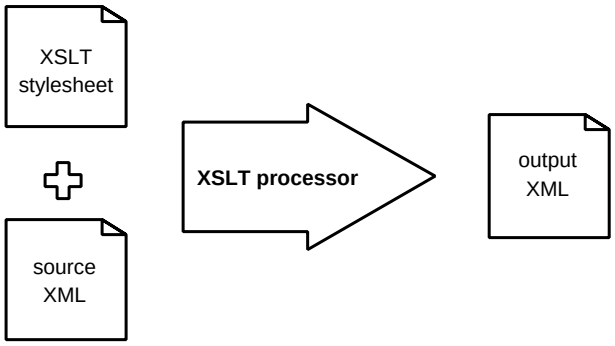

Figure 2: XSLT transformation

The transformation of the conceptual model to the visual model (shown in Figure 1) is not necessarily a single step process. The pipeline architecture allows an arbitrary series of transformations. This makes it possible to decompose complex transformations into a series of simpler ones and to reuse transformations common to several visualisations.

XSLT is suitable for a wide class of transformations, but it is not a general purpose programming language. Some elaborate transformations are better implemented in a conventional language. Support for reading and writing XML from conventional languages is readily available, so that a custom tool can be developed and inserted into the pipeline.

The pipeline as shown in Figure 1 is ideal for visualisations that depend only on the syntactic analysis of source code. More sophisticated visualisations are likely to also require semantic analysis of the code. A semantic analyser can be incorporated into the parser, or as a more general solution, inserted into the pipeline as the first transformation of the conceptual model. Figure 3 shows a more involved pipeline, including a pipelined semantic analyser and XSLT transformations driven by stylesheets.

Parsing, transformation and visualisation are addressed in more detail in the following sections.

4 Parsing

Source code enters the visualisation pipeline by being parsed. The extracted information is made available to downstream tools as an XML file. Thus, the applicability of our visualisation pipeline is limited to those programming languages for which we have parsers that emit XML.

Although parsers exist for all programming languages, high quality parsers designed to expose comprehensive parse information are rare. Some commercial and academic parsers are intended for this purpose, but lack the flexibility and rigour we desire for a general-purpose visualisation pipeline. Existing parsing tools are sometimes restrictive in exposing parse information and often are unable to cope with the variety of dialects and complex syntactic combinations found in the large-scale industrial software we wish to visualise. Further, any one parser is limited to a particular programming language (or set of languages). A single technology applicable to all programming languages would be preferable to a diverse set of parsing tools. Consequently, we have adopted the approach of generating parsers for the languages of interest.

The syntax of a programming language is defined by a grammar—a set of rules by which sentences in that language may be composed. A parser generator is a tool that will accept a grammar and automatically generate a parser. The parser can subsequently be used to recognize any sentence in that language. The archetypal parser generator is yacc (Johnson 1975), but many alternative tools exist.

A parser generator produces a parser that uses a particular parsing approach. The power of the approach dictates the class of grammars that the parser generator can accept. Some parsing approaches are powerful enough to handle any Context Free grammar, but in practice are unsuitable for parsing source code because they parse long sentences too slowly. The best known general parser is perhaps that of Earley (Earley 1970), which exhibits $O(n^2)$ time complexity for non-ambiguous grammars, where n is the number of tokens in the sentence. For ambiguous grammars, performance degrades to $O(n^3)$. Other general grammars, such as those of Unger and CYK, do no better (Grune & Jacobs 1990). Consequently, mainstream parser generators usually adopt weaker parsing approaches that accept only a subset of Context Free grammars, but provide linear performance. The dominant approaches for parsing programming languages are LL(k) and LALR(1).

If an LL or LALR parser generator is given a grammar that is not LL or LALR respectively, then the parser generator fails to produce a parser for that language. This means that in order to build a programming language parser using a parser generator, it is often necessary to modify the grammar to satisfy the restrictions imposed by the parsing technology.

Many programming language grammars can be made to satisfy the constraints of these weaker parsing approaches without changing the underlying set of sentences the parser will accept. However, for the purpose of visualising and measuring software, grammar modifications are undesirable. A program should be measured and visualised in terms of a standard definition of the language—its standard grammar. The standard grammar of a particular language may not have been crafted for the parsing technology in question, and so may require extensive modification, thus reducing the value of measurements and visualisations.

More profoundly, some programming languages—C++ in particular—fall so far outside the capabilities of these conventional parsing approaches that modifying the grammar is impractical. In an earlier paper (Irwin & Churcher 2001), we discussed the difficulty of using conventional parser generators for C++ and identified a solution: Tomita parsing, a stronger parsing approach developed for natural languages (Tomita 1986). Tomita parsing can handle all but the most ambiguous grammars in linear (or near-linear) time. We demonstrated that it greatly simplifies the task of parsing C++.

Tomita parsing extends LR parsing by tolerating ambiguity in the parsing automaton. It replaces the stack used in a Push Down Automaton (PDA) with a graph, allowing ambiguous paths to be pursued until non-viable ones are eliminated. This stronger approach can accept any grammar without modification. It does so without significantly increasing the order of time complexity of the parse, for any realistic programming language grammar (whereas pathological grammars can degrade to exponential time complexity) (Grune & Jacobs 1990).

With Tomita parsing, we have the capability to mechanically produce a parser for any standard programming language grammar. Because some programming languages do not require this level of power, an ideal parser generator would use an appropriately powerful parsing approach for each grammar. We have built such a tool, using XML to separate the concerns of the parser generation process and gain

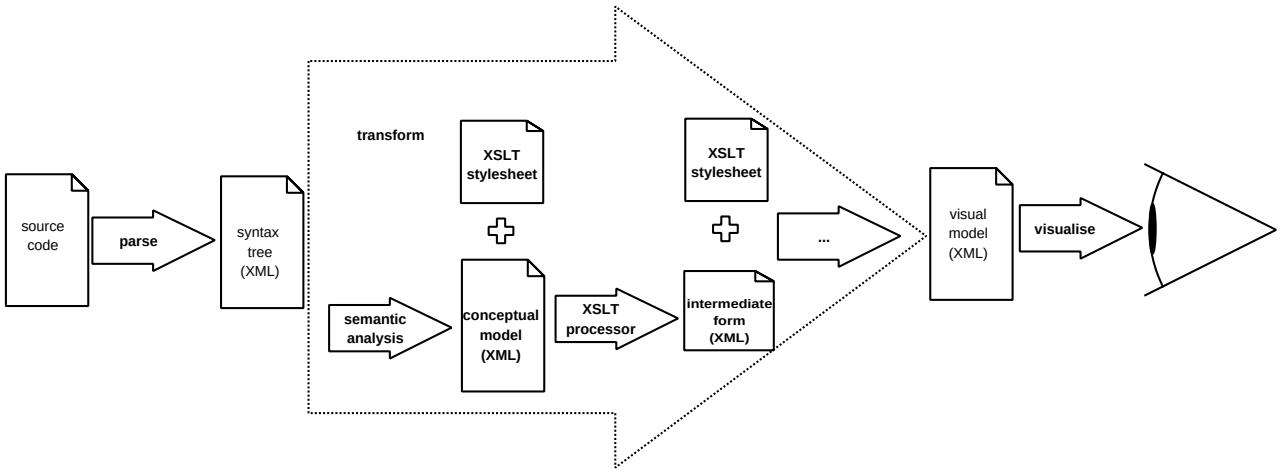

Figure 3: A more complex pipeline

flexibility.

4.1 Yakyacc

Yakyacc (yet another kind of yacc) is a parser generator that accepts a grammar in Extended Backus Naur Form (EBNF) and constructs a series of successively more powerful LR parsers until an adequate parser is attained. Figure 4 shows the parsing approaches used by yakyacc, with an automaton of the weakest class, LR(0), being produced first, and then upgraded to SLR(1), LALR(1), LR(1) and finally Tomita parsing.

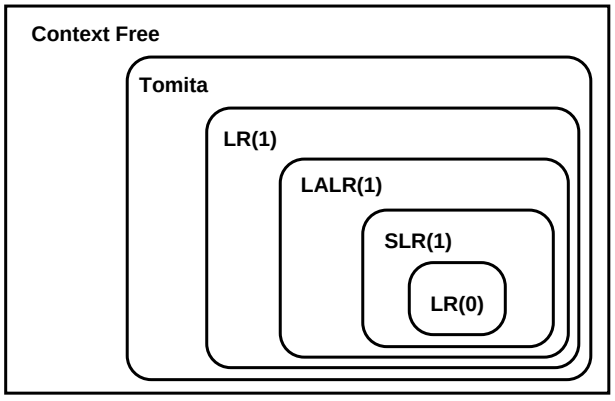

Figure 4: Grammars currently supported by yakyacc

Most parser generators, including yacc and its ilk, directly emit a program that can be compiled to an executable parser. While convenient for some purposes, this approach forces the user to accept a parser written in the language (or languages) supported by the parser generator. It also requires the user to specify the runtime behaviour of the parser in conjunction with the specification of the grammar; most parser generators do this by allowing action code to be embedded in the grammar specification. If, for example, a yacc-based parser is to emit an XML description of the parse tree, then code to build and output the tree must be embedded in the grammar.

For the purposes of a visualisation pipeline, embedding actions in a grammar is awkward. We want the freedom to change the output constructs and level of detail independently of the grammar, and even to maintain a suite of output configurations for one grammar. Figure 5 shows the architecture of yakyacc, which uses an intermediate XML file containing a description of the parsing PDA. This XML file separates the task of processing the grammar into a PDA from the task of generating that PDA as a parser program.

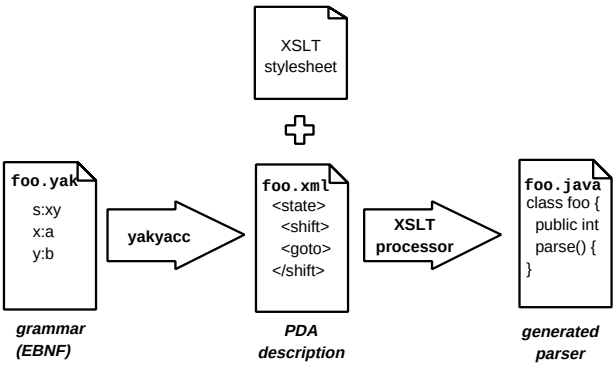

Figure 5: Using `yakyacc` to generate a parser for Java

Yakyacc produces an XML file containing a marked-up description of the original grammar and a PDA capable of parsing that grammar. The power of the PDA is, by default, the least powerful parsing approach necessary, but may be more or less powerful as requested by the user. If Tomita parsing is necessary, the PDA defaults to LR(1) with some conflicts remaining.

Code generation is accomplished by using XSLT to transform the PDA into code. By changing the stylesheet used, the language in which the parser is written and the actions it performs may be changed. Thus, for any one PDA, it is possible to use one stylesheet that emits a Java-based parser and another that emits a C-based parser, for example. The first parser may output an entire syntax tree, while the second might output only high-level constructs.

A user of yakyacc, at the price of having to know how to write a stylesheet, can customise the parser generation arbitrarily. Even without knowledge of XSLT, a user could choose from a set of predefined transformations, attaining greater flexibility than is available with other tools. One such predefined stylesheet produces a Tomita parser by generating a PDA that uses a graph in place of a stack; another produces a conventional PDA. The user is free to choose either stylesheet after learning from yakyacc what level is required to guarantee complete parsing.

Apart from the stack, Tomita parsing relies on a conventional parsing automaton. Toleration of nondeterminism works with any class of LR parser—

```
public class HelloWorld {
    private int i;
    public static void main(String args[]) {
        System.out.println("Hello world");
    }
}
```

Figure 6: A tiny Java program

the weaker the approach, the more branching in the Tomita machine. Yakyacc's two-stage generation allows any level of PDA to supply the finite state machine for the Tomita parser. So, for example, we can parse C++ with a Tomita parser containing either an SLR(1) machine, or a full LR(1) machine. The latter will be slightly faster. Java can be parsed with a deterministic LALR(1) machine, or a Tomita parser containing a non-deterministic LR(0) machine.

As an example, we built a Java parser that emits a complete Java parse tree in XML. We used a publicly available Java 1.2 BNF grammar from another parsing tool, Java CUP (Appel 1998). This grammar was free of conflicts at LALR(1), so yakyacc produced a conventional LALR(1) PDA, described in XML. We then wrote a stylesheet to transform the PDA, yielding a Java class capable of parsing Java. Finally, this class was compiled and executed in conjunction with a Java scanner, also appropriated from Java CUP.

Parse trees make the structure of source code explicit, so are naturally much more verbose than the original code. Parsing the minimal Java HelloWorld program of Figure 6 produces an XML parse tree containing 169 tags. A fragment is shown in Figure 7.

```
...
<class_declaration>
   <modifiers_opt>
   </modifiers_opt>
   <token id='CLASS'>class</token>
   <token id='IDENTIFIER'>HelloWorld</token>
   <super_opt>
   </super_opt>
   <interfaces_opt>
   </interfaces_opt>
   <class_body>
     <token id='LBRACE'>{</token>
     ...
```

Figure 7: Part of the parse tree for the program of Figure 6

This Java parser describes the parse tree in XML using one tag type for each non-terminal in the grammar. Tokens are described with the `<token>` tag, which contains an attribute identifying the kind of token, and has as its value the original source code. In this way, the entire source code remains (with the exception of whitespace), embedded in the markup of the syntax tree.

Because this XML output exposes complete parse information and retains the original source, it is suitable for a wide variety of visualisations. Downstream transformations can filter the parse information to a subset relevant to specific visualisations. More elaborate transformations can combine the information from one source file with others to enable semantic checks, cross-referencing and larger-scale visualisations.

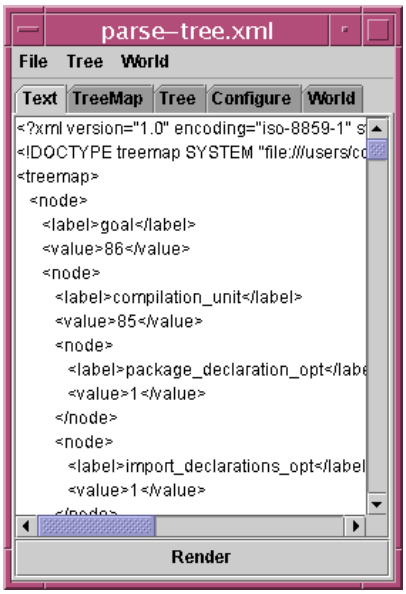

Figure 8: XML input

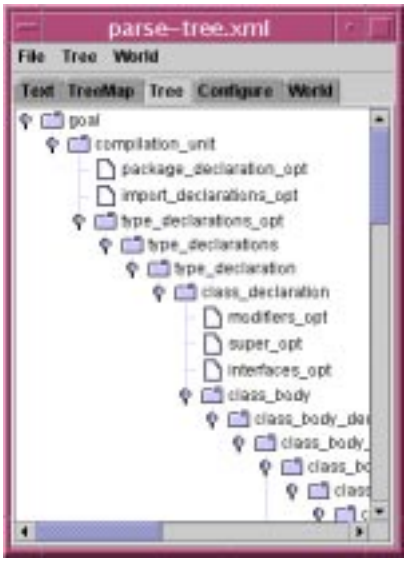

Figure 9: Conventional tree

Figure 10: VRML generation

5 Transformations

The transformation phase of the visualisation pipeline maps a conceptual model to a visualisation model. This is not necessarily a mechanical one-to-one conversion. Transformations can filter information, aggregate details into higher-level abstractions and make implicit information explicit and vice versa.

Transformation actions can be decomposed into any number of steps, with the intermediate forms of the data being retained. This record of the progress of the pipeline is valuable for developing and experimenting with visualisations; any stage can be revisited to modify its behaviour or add branches to the pipeline.

The human-readable nature of XML is a significant advantage in this setting. A pipeline developer can readily observe the effects of transformations, making it easier to observe progress and debug the pipeline. If necessary, the files can be manually edited; this is helpful for debugging and for one-off experiments where automating the process is not worthwhile.

The use of XML in the visualisation pipeline provides the advantages of having explicit metadata in the data files. This means that any intermediate file is a comprehensible entity in its own right; it is self-describing, so can be understood without intimate knowledge of the tools that created it. The alternative is to have files in formats defined elsewhere, often with definitions obscured in the tools that manipulate the files. XML enables much greater freedom to bolt on new transformations without knowledge of, or coupling to, other tools in the pipeline.

XSLT adds further advantages. It is a powerful mechanism for specifying transformation and encapsulates transformations in a stylesheet file that can be reused and modified. It is designed to inter-operate with XML and is robust and flexible in the face of modifications to XML input files.

We have developed a number of XSLT stylesheets to test our pipeline approach and some of these are discussed in the following section.

6 Visualisation: Some Applications and Results

In this section we present some results obtained from tools we have built using the XML-based architecture we have described. Our intention is to demonstrate the feasibility and inherent flexibility of our approach.

A large number of visualisation techniques are available (Spence 2001, Stasko, Domingue, Brown & Price 1998, for example) and we make use of a number of these in our software visualisation research.

Treemaps (Johnson & Shneiderman 1991) are one technique we use extensively. Figures 8–11 show our XML-based treemap application, tmtool, in action. As shown in Figure 8, an XML description of a treemap may be entered or (more realistically) edited. Other representations, such as conventional tree layout (Figure 9) are available and the application generates a VRML description (Figure 10) according to the configuration options chosen by the user.

Figure 11 shows tmtool's 2D display of an XML file representing the parse tree resulting from running our instrumented Java parser on the program listed in Figure 6. One of the stylesheets mentioned in Section 5 converts the XML parse tree output of the Java parser directly into the format expected by tmtool.

Figure 12 shows a view of a VRML world containing a nested treemap—whose source code is that of Figure 10, which in turn is derived from the data obtained by parsing the program listed in Figure 6.

Another of our XSLT stylesheets transforms the parse tree data into the XML format expected by Angle (Churcher & Creek 2001), our 3D graph layout tool. Angle can generate VRML worlds containing the 3D drawing of the input graph. The VRML world shown in Figure 13 contains the same parse tree data as Figure 11 rendered as a 3D tree.

Figures 14 and 15 show the results of applying other XSLT stylesheets to generate VRML worlds containing visualisations of further aspects of our system.

Figure 14 shows a visualisation of the coupling between methods for a Java class. This is the basis for the LCOM metric (Chidamber & Kemerer 1994). The properties of the class (green cubes) are connected (by yellow edges) to the (magenta cylinder) representing the class. The remaining edges indicate methods (red spheres) which access properties. The sea-urchin-like spiny balls at the left and right edges of the figure suggest that there are two properties which are rarely accessed by the same methods, indicating a possible low cohesion.

A final example is shown in Figure 15 which shows a visualisation of the a call graph for methods within the same Java class. Features such as long chains and fan-in/fan-out are clearly visible.

It is difficult to describe adequately on paper the features of our VRML worlds and we encourage readers to visit our web site (`http://www.cosc.canterbury.ac.nz/research/RG/svg`) to experience the worlds for themselves.

7 Conclusions

We have demonstrated how the conventional visualisation pipeline concept may be expanded. XML allows a true pipeline consisting of individual products (documents) and processes (transformations) to be constructed. Our experience with yakyacc, tmtool, Angle and our other software visualisation tools indicates that this approach is feasible and that it has significant advantages. These include increased flexibility (only a little "plumbing" is needed to modify the pipeline), potential for increased rigour (exposing the details of intermediate artifacts enables validation and comparison of empirical results) and the ability to visualise the pipeline itself.

We intend to continue to explore the applications of XML technology in our software visualisation research. XML is a somewhat verbose language and our challenges in the immediate future include ways of reducing the size of the files we must process. Our medium-term goal is the integration of our existing tools with a wide range of input sources and an equally wide range of visualisation formats.

References

Appel, A. (1998), *Modern Compiler Implementation in Java*, Cambridge University Press.

Carey, R. & Bell, G. (1997), *The Annotated VRML 2.0 Reference manual*, Addison-Wesley.

Chidamber, S. & Kemerer, C. (1994), 'A metrics suite for object oriented design', *IEEE Transactions on Software Engineering* **20**(6), 476–493.

Churcher, N. & Creek, A. (2001), Building virtual worlds with the big-bang model, *in* 'invis.au: Australian Symposium on Information Visualisation', Sydney, Australia. to appear.

Churcher, N., Keown, L. & Irwin, W. (1999), Virtual worlds for software visualisation, *in* A. Quigley,

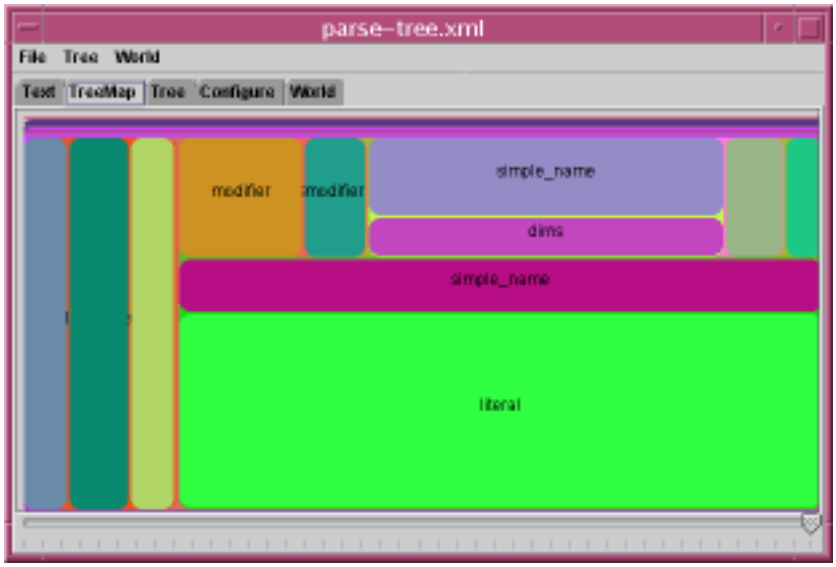

Figure 11: 2D treemap representation

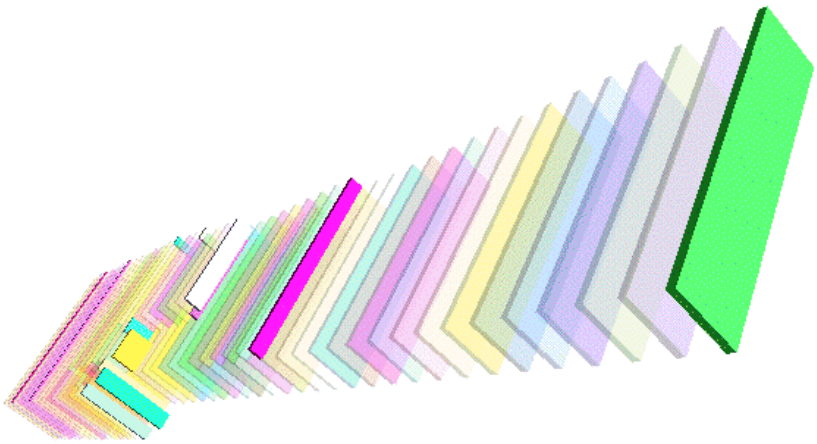

Figure 12: Parse tree represented as treemap

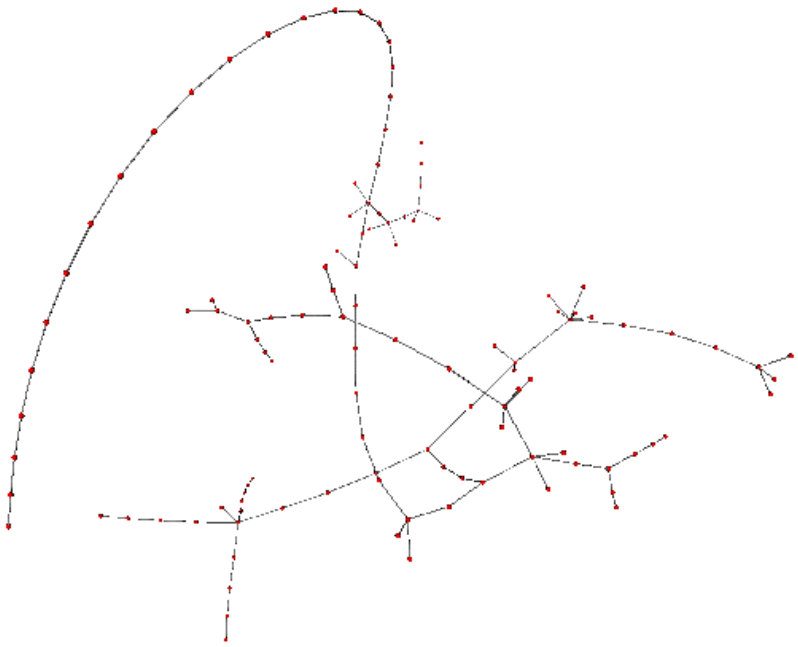

Figure 13: Parse tree represented as 3D tree

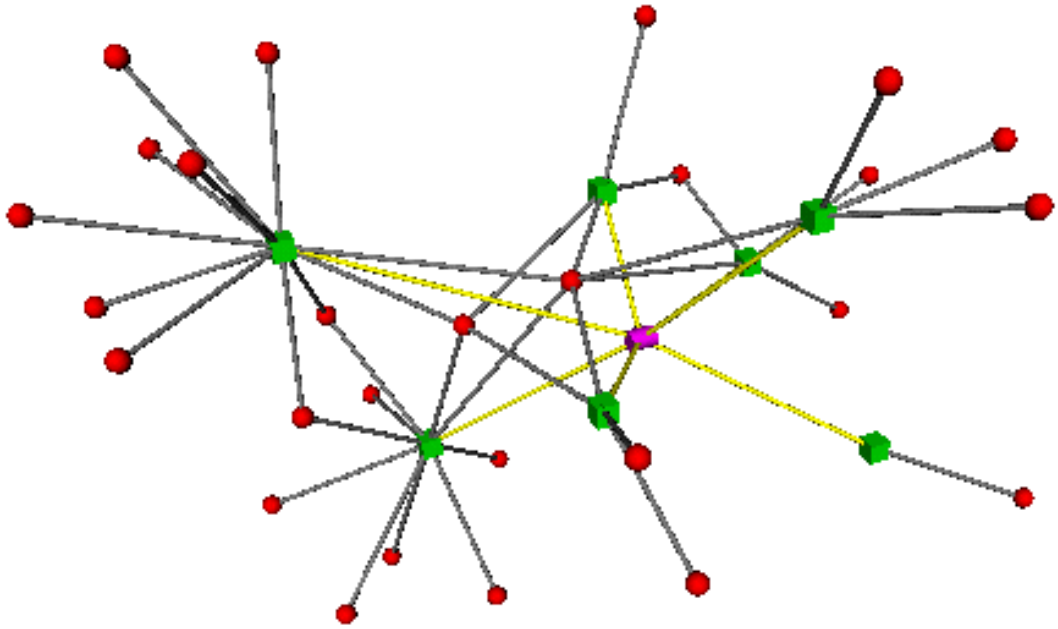

Figure 14: A representation of the cohesion of a Java class

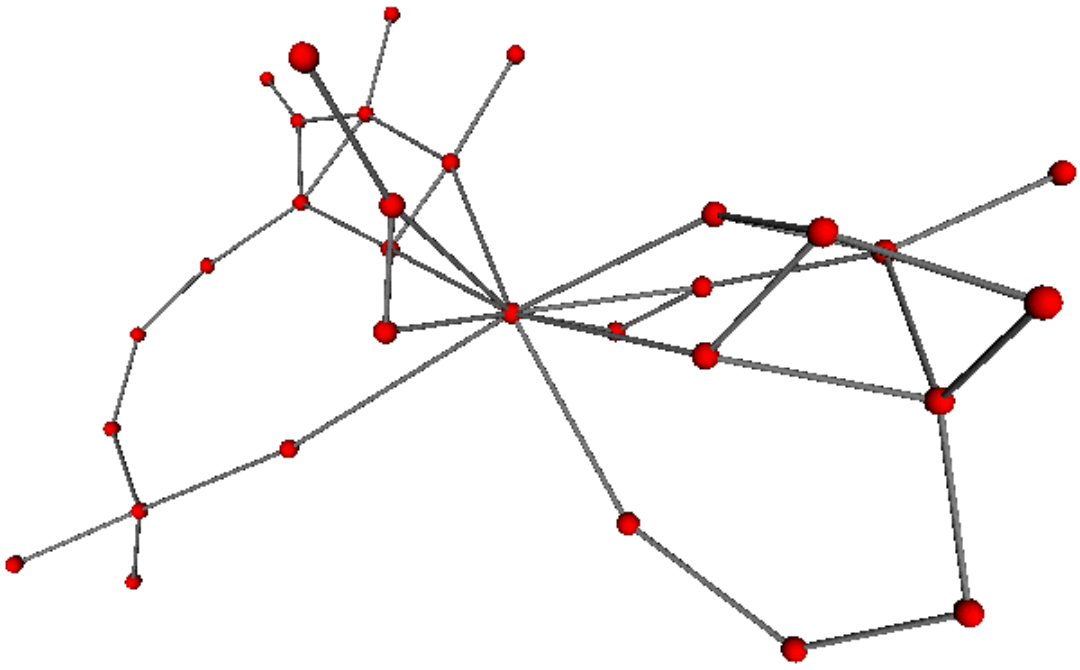

Figure 15: A call graph for the class whose cohesion is represented in Figure 14

ed., 'SoftVis99 Software Visualisation Workshop', University of Technology, Sydney, Australia, pp. 9–16.

Earley, J. (1970), 'An efficient context-free parsing algorithm', *Communications of the ACM* **13**(2), 94–102.

Griswold, W. (2001), Making slicing practical: The final mile, *in* 'PASTE'01: ACM SIGSOFT–SIGSOFT Workshop on Program Analysis for Software Tools and Engineering', ACM Press, Snowbird, Utah, p. 1.

Grune, D. & Jacobs, C. (1990), *Parsing Techniques: A Practical Guide*, Ellis Horwood.

Hartley, D., Churcher, N. & Albertson, G. (2000), Virtual worlds for web site visualisation, *in* 'Proc APSEC2000, 7th Asia Pacific Software Engineering Conference', IEEE Press, Singapore, pp. 448–455.

Irwin, W. & Churcher, N. (2001), 'A generated parser of c++', *N.Z. Journal of Computing* **8**(3), 26–37.

Johnson, B. & Shneiderman, B. (1991), Tree-maps: A space-filling approach to the visualization of hierarchical information structures, *in* G. Nielson & L. Rosenblum, eds, 'proc. Visialization '91', IEEE Computer Society Press, Los Alamitos, CA, pp. 284–291.

Johnson, S. C. (1975), Yacc, yet another compiler compiler, Computing Science Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ.

Martin, D., Birkbeck, M., Kay, M., Loesgen, B., Pinnock, J., Livingstone, S., P.Stark, Williams, K., Anderson, R., Mohr, S., Baliles, D., Peat, B. & Ozu, N. (2000), *Professional XML*, Wrox Press.

Schroeder, W., Martin, K. & Lorensen, B. (1998), *The Visualization Toolkit: An Object-Oriented Approach to 3D Graphics*, 2nd edn, Prentice Hall.

Spence, R. (2001), *Information Visualisation*, Addison-Wesley.

Stasko, J., Domingue, J., Brown, M. & Price, B., eds (1998), *Software Visualization: Programming as a Multimedia Experience*, MIT Press.

Tomita, M. (1986), *Efficient Parsing for Natural Language*, Kluwer Academic Publishers.

XML (2001), 'XMLORG', http://www.xml.org.