

Working Session: Textual Views of Source Code to Support Comprehension

Anthony Cox
Faculty of Computer Science
Dalhousie University
Halifax, Nova Scotia
amcox@cs.dal.ca

Michael L. Collard
Department of Computer Science
Kent State University
Kent, Ohio
collard@cs.kent.edu

Abstract

Source code can be viewed in many ways, with each view facilitating access to different information contained within the code. In this working session, we will explore the role that marked-up textual views can provide in support of software comprehension and maintenance. Text has the advantages of being easily communicated, effectively manipulated with existing tools, and highly scalable. Furthermore, marked-up text models may improve comprehension by expressing information directly within the context of maintainers' focus – the source code they are manipulating. This session intends to explore the expressibility of marked-up text and its applicability in support of program comprehension tasks. Topics will include: the roles these models fulfill, their limitations, their combination, and the exploration of future research directions.

1 Theme

There are many perspectives that maintainers can use when viewing source code. Code can be considered as a graph based on control or data flows, as a tree based on parsing rules or abstract syntax, or as text, stored in text files and manipulated with a text editor. Each perspective provides different information to maintainers, and while useful, is not sufficient on its own to support all maintenance tasks.

However, when supplemented with other representation techniques, the textual model can be used as a storage and exchange format for the other views. The addition of markup to source code permits the code to be enhanced with graphical and tree structured information. In the past few years, many approaches for combining markup and source code have been examined (e.g., [13], [1], [14], [19], [3], [10], [11]).

Source code modeling and applications using marked-up textual models have been regular and recent topics at IWPC (e.g., [8], [17], [4], [7]), indicating that the topic is of interest from a research perspective. As well, recent keynote

addresses at IWPC have stressed the importance of preserving the programmer's view of source code (which textual views directly support) [6] and the differences (at the syntax level) between program understanding and compiling [15]. A review of the papers being presented at IWPC'05 indicates that marked-up textual models are still of interest to the research community.

In earlier work at IWPC, Tilley and Smith suggested that HTML provides an effective infrastructure for reengineering [22]. The dramatic growth of the WWW and the development of XML indicates that marked-up textual models are highly topical and have the potential to significantly impact research.

The use of textual markup models provides a combination of advantages that other models (e.g., [24]) do not possess. These advantages are:

- **Robustness:** Text models can be used when other models are not extractable, such as when source code can not be parsed effectively.
- **Scalability:** Large scale text data-sets can be efficiently stored and retrieved using text-oriented databases (e.g., Google) and processed as memory-efficient streams (e.g., SAX).
- **Searchability:** Text is easily searched using string-based or index-based tools (e.g., `grep`¹ or Google).
- **Independence:** Tools for manipulating text (e.g., Perl) or marked-up text (e.g., XSLT) can be used on any textually represented programming language.
- **Adoption-Centric:** Text manipulation tools already exist and are regularly used for a variety of tasks (e.g., Perl, AWK, `grep`, `vi`, `emacs`).
- **Readability:** Text is always readable by maintainers, potentially increasing maintainers' trust and understanding.

¹`grep` is maintainers' most commonly used search tool [21].

- Communicable: Text is easily communicated between tools, hosts, and environments.
- Transparency: The relationship between extracted information and the source code is easily seen without the need to build and maintain an external mapping.
- Abstraction: Textual representations can support many different levels of abstraction. For example, text can support syntactic structure [17] or function calling behaviour [14].

While one can certainly argue the merits of a textual representation and the importance of the ‘advantages’ that are listed here, there is a large body of literature on the use of marked-up textual models. Some of this literature is now presented to demonstrate that the approach is effective and useful. We examine the use of markup in support of source code generation, transformation, browsing, visualization and tool interoperability (exchange).

Programmers are beginning to see markup integrated into source code. For program development, XAML² permits programmers to specify visual interfaces using a combination of text and markup. After creation, XAML files are pre-compiled to generate compilable source code. XBL and XUL fulfill a similar role in describing visual interfaces and behaviour.

To support source code transformation, the results of analyses have been represented using an “ultra-high level executable specification language” called HSML [8]. HSML demonstrates the scalability of markup approaches since it has been shown to be 10% faster than the LS/2000 system used to process over 3 billion lines of code for year 2000 compatibility. Collard and Maletic have examined the use of XML to support a source code transformation approach to refactoring [5]. Changes to source code files have been identified using an XML-based format (srcML) and subsequently recorded in a complementary format (srcDiff) [16].

To facilitate browsing of source code, tools such as CToHTML³ permit code to be examined using a generic web-browser. Devanbu *et al.* created a framework for inserting HTML links into software documents (source code) so that WWW-based browsing can be added to existing development and maintenance environments [13].

Cowan *et al.* used SGML to record syntactic and semantic information in C programs [9]. Software visualization tools then used this markup to control the appearance (e.g., size, colour, font) of program constructs, and consequently, to enhance the code’s comprehensibility and readability. Cross and Hendrix created GRASP-ML to repre-

sent control-flow constructs in Ada [12]. In their research, markup captured and stored the information needed to create a static control structure diagram.

Markup has been widely explored as a means of exchanging language between tools. Mamas and Kontogianis used XML derived markup languages (JavaML and CppML) to record the abstract syntax of C++ and Java [19]. These XML representations are then used to facilitate integration of the tools in the Integrated Software Maintenance Environment through the Document Object Model (DOM) interface for accessing XML files. Similarly, Boshernitsan and Graham examined the use of XML for encoding abstract syntax and exchanging it between tools in the Harmonia framework [3]. To enhance communication between tools, the *Workshop on Software Exchange Formats* [20] identified GXL [14] – an XML-based representation – as the most promising format.

While there are other projects which we have not mentioned, the projects listed here demonstrate the flexibility and wide-spread use of markup models for source code. Thus, we have evidence that programmers use markup-based textual models during maintenance and development and that these models are effective for a variety of tasks.

2 Perspectives

While there is agreement that markup models are useful, the properties of these models can differ significantly, as the following examples demonstrate:

- Structure: hierarchical (e.g., CppML [19]) versus non-hierarchical models (e.g., Maia [10]).
- Concurrency: single (e.g., XML) versus multiple concurrent hierarchies (e.g., SGML).
- Focus: representational with code embedded in the representation (e.g., JavaML [1]) versus code-oriented with the representation embedded in the code (e.g., srcML [17, 4]).

In part, the various properties of source code models can be considered as a consequence of the variation in perspectives of software. One can take a textual view and represent information within the source code [8]. Alternatively, one can focus on syntax and create a representation based on the code’s abstract syntax tree (AST) [19]. Finally, one can use an even more abstract view and consider software as a graph [14]. These perspectives, text, tree and graph, can all be represented using markup.

It has been argued that the representation of the documentary structure of source code (i.e., the structure of documentary elements such as whitespace and line-structure) is a continuing problem for language-based source code representations [23]. This problem illustrates the difficulty in

²See <http://msdn.microsoft.com/msdnmag/issues/04/01/Avalon/default.aspx>

³Available from <http://www.cs.washington.edu/homes/zahorjan/homepage/Tools/CToHTML/ctohtml.htm>.

creating a uniform representation for orthogonal kinds of information. However, Van De Vanter suggests that the Legasys approach [18] preserves documentary structure and provides a potential solution through the direct mapping of maintenance activities to source text.

As well as the loss of documentary structure, many representations do not preserve syntactic alternatives for constructs. For example, AST representations do not preserve declaration style and literal formatting choices. This information may be needed to support comprehension activities such as chunking. One approach that preserves the original source code structure is the DMS system of Baxter which enhances an abstract syntax graph with textual data so that it is preserved and available [2].

To access a software model it is possible to use a tool's Abstract Programming Interface (API). However, textual interfaces provide a much richer view of the source code. Text is a representational format while APIs (which are typically object-oriented) are programmatic abstractions for accessing source code. Textual representations can be mapped to an API view that fits the task at hand. This technique is much less restrictive than using a singular in-memory graph or tree view. As such, textual formats permit a wide range of views and can be used with a wide range of tools. For example, a marked-up document can be viewed as a sequence of characters (using regular expressions), an in-memory object (using DOM), or as a stream of parsing events (using SAX). In addition, pattern matching can be used that crosscuts an object-oriented view of the source code. Thus, textual formats demonstrate strong flexibility as well as a powerful representational capacity.

Adding markup to text permits a schema language (e.g., DTD, XSchema, RelaxNG) to be used. The schema may be limited to restricting the markup elements and their attributes, or may be more broad in scope and used to restrict the text as well. A strict schema can prevent the representation of syntactically incorrect code, but has difficulty preventing the representation of semantically incorrect code (i.e., C with undeclared variables). Less strict schemas will allow the representation of uncompileable code (e.g., C with preprocessor constructs) or incomplete code fragments. While each approach has merit, the end choice is constrained by the schema language used and the desired application.

3 Goals

The goal of this working session is to explore the application of textual models in support of program comprehension and maintenance. This exploration is intended to examine current research philosophies, application roles, model properties and implementation details. The session is intended to answer questions such as:

1. When and where are marked-up textual models appropriate and effective? How are more problematic issues (e.g., whitespace, comments, preprocessor constructs) handled?
2. Are there agreed upon models that adequately represent extracted information across a range of abstraction levels?
3. How are issues such as version control and project management supported? Can these formats support multiple versions and the differences between them? If not, can they be extended?
4. Is it possible to map extracted information into source code? For which levels of abstraction is this effective?
5. While XML is well supported and many markup models are XML-compliant, do the limitations of XML hinder its utility?
6. Is a single model sufficient, or should several models be combined? If so, how should they be combined (e.g., layers)?
7. Is it possible to define a 'most-general' model that can be easily transformed into other models?
8. What is the role of the schema for these representations? What part of the representation should they constrain? Should they be restricted to allow only the representation of compilable code?
9. What have past experiences taught us and where should we be going?

By answering these questions, the session will provide a foundation for future research in this area while beginning a dialog between the participants. As well, we hope to explore avenues for future research and collaboration between participants.

4 Activities

Participants will be asked to provide a 2–3 page position paper. A set of brief presentations will begin the session. Participants will be given 5 to 7 minutes to present their position. The position papers will be made available to presenters in advance to permit comparison and contrast in the presentations. A discussion and question period will follow the presentations. Questions will not be permitted after each presentation in order to keep the presentations occurring in a timely manner. This restriction is also intended to create a more lively and engaged discussion period. Upon completion of the session, the organizers will create a report containing the position papers and a summary of the presentations and discussions. This report will be made publicly available through an appropriate means.

References

- [1] G. Badros. JavaML: A markup language for Java source code. *Computer Networks*, 33(1-6):159–177, June 2000.
- [2] I. Baxter, C. Pidgeon, and M. Mehlich. DMS: Program transformations for practical scalable software evolution. In *International Conference on Software Engineering*, pages 625–634, Edinburgh, Scotland, May 2004.
- [3] M. Boshernitsan and S. Graham. Designing an XML-based exchange format for harmonia. In *Seventh Working Conference on Reverse Engineering*, pages 287–289, Brisbane, Australia, November 2000.
- [4] M. Collard, H. Kagdi, and J. Maletic. An XML-based lightweight C++ fact extractor. In *International Workshop on Program Comprehension*, pages 134–143, Portland, OR, May 2003.
- [5] M. Collard and J. Maletic. Document-oriented source code transformation using XML. In *First International Workshop on Software Evolution Transformation*, pages 11–14, Delft, Netherlands, November 2004.
- [6] J. Cordy. Comprehending reality – Practical barriers to industrial adoption of software maintenance automation. In *International Workshop on Program Comprehension*, pages 196–205, Portland, OR, May 2003.
- [7] J. Cordy. Generalized selective XML markup of source-code using agile parsing. In *International Workshop on Program Comprehension*, pages 144–153, Portland, OR, May 2003.
- [8] J. Cordy, K. Schneider, T. Dean, and A. Malton. HSML: Design directed source code hot spots. In *International Workshop on Program Comprehension*, pages 145–154, Toronto, Canada, May 2001.
- [9] D. Cowan, D. Germán, C. P. de Lucena, and A. von Staa. Enhancing code for readability and comprehension using SGML. In *International Conference on Software Maintenance*, pages 181–190, Victoria, Canada, September 1994.
- [10] A. Cox and C. Clarke. Representing and accessing extracted information. In *International Conference on Software Maintenance*, pages 12–21, Florence, Italy, November 2001.
- [11] A. Cox and C. Clarke. Three-layered source-code modelling. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 91:71–79, May 2004. Issue devoted to Proceedings of the Workshop on Meta-Models and Schemas for Reverse Engineering, Victoria, Canada, November 2003.
- [12] J. Cross and T. Hendrix. Using generalized markup and SGML for reverse engineering graphical representations of software. In *Second Working Conference on Reverse Engineering*, pages 2–6, Toronto, Ontario, Canada, July 1995.
- [13] P. Devanbu, Y.-F. Chen, E. Gansner, H. Müller, and J. Martin. CHIME: Customizable hyperlink insertion and maintenance engine for software engineering environments. In *21st International Conference on Software Engineering*, pages 473–482, Los Angeles, California, May 1999.
- [14] R. Holt and A. Winter. A short introduction to the GXL software exchange format. In *Seventh Working Conference on Reverse Engineering*, pages 299–302, Brisbane, Australia, November 2000.
- [15] P. Klint. How understanding and restructuring differ from compiling – A rewriting perspective. In *International Workshop on Program Comprehension*, pages 2–11, Portland, OR, May 2003.
- [16] J. Maletic and M. Collard. Supporting source code difference analysis. In *International Conference on Software Maintenance*, pages 11–17, Chicago, IL, September 2004.
- [17] J. Maletic, M. Collard, and A. Marcus. Source code files as structured documents. In *Tenth International Workshop on Program Comprehension*, pages 289–292, Paris, France, June 2002.
- [18] A. Malton, K. Schneider, J. Cordy, T. Dean, D. Cousineau, and J. Reynolds. Processing software source text in automated design recovery and transformation. In *Ninth International Workshop on Program Comprehension*, pages 127–134, Toronto, Canada, May 2001.
- [19] E. Mamas and K. Kontogiannis. Towards portable source code representations using XML. In *Seventh Working Conference on Reverse Engineering*, pages 172–182, Brisbane, Australia, November 2000.
- [20] S. Sim and R. Koschke. WoSEF: Workshop on standard exchange format. *Software Engineering Notes*, 26(1):44–49, January 2001.
- [21] J. Singer and T. Lethbridge. What’s so great about ‘grep’? implications for program comprehension tools, 1997. Available as: <http://wwwsel.iit.nrc.ca/singer/grep/greptxt.html>.
- [22] S. Tilley and D. Smith. On using the web as infrastructure for reengineering. In *Fifth International Workshop on Program Comprehension*, pages 170–173, Dearborn, Michigan, May 1997.
- [23] M. Van De Vanter. The documentary structure of source code. *Information and Software Technology*, 44(13):767–782, October 2002.
- [24] K. Wong and J. Uhl. *Rigi User’s Manual, Version 5.4.4*. The Rigi Group, Victoria, Canada, June 1998.