Using Generalized Markup and SGML for Reverse Engineering Graphical Representations of Software

James H. Cross and T. Dean Hendrix
Computer Science and Engineering
Auburn University
Auburn, AL 36849
cross@eng.auburn.edu, thendrix@eng.auburn.edu

Abstract

As part of the ongoing research of Auburn University's GRASP Project (Graphical Representations of Algorithms, Structures, and Processes), a markup language has been designed and prototyped to facilitate the automatic generation of static program visualizations from source code. Specifically, the latest release of the GRASP/Ada tool uses a markup language called GRASP-ML as the basis for automatically generating control structure diagrams from Ada source code. This markup language is described and its role in reverse engineering with GRASP/Ada is explained. Finally, promising future work is outlined and discussed.

1 Introduction

A primary goal of reverse engineering is to support reusability, verification, and maintenance of software. For this process to be successful, a sufficient design-level understanding of the software must be attained. This usually involves creating representations of the software in another form at the same relative level of abstraction or at a higher level of abstraction [1]. These alternate forms and representations may be goals in and of themselves, as in a redocumentation effort, or as comprehension aids for subsequent reverse engineering or reengineering efforts.

A primary goal of our current work is to demonstrate that a *markup language* can be used to capture information necessary for creating graphical representations and abstractions in a generalized and persistent manner, and in so doing yield significant benefits. GRASP-ML is a markup language designed by the authors to capture information necessary to automatically generate static program visualizations from source code or PDL. We employ GRASP-ML as the basis of the latest release of GRASP/Ada, a program visualization tool for reverse engineering control structure diagrams from Ada source code [2]. The following sections describe the basis of our efforts, the structure and design of GRASP-ML, how GRASP-ML is used to facilitate reverse engineering control structure diagrams, and directions for future research.

2 Graphical Representations of Source Code

Graphical representations have been recognized as playing an important role in communication and understanding from the perspective of both the writer and the reader. Graphical representations can have a positive impact in communicating throughout the stages of design, implementation, testing, and maintenance. Considering the resources expended in the software development life cycle, effective graphical representations which aid communication and comprehension could significantly reduce software project costs. Indeed, [3] found that code reading was the most effective method of detective errors during the verification process when compared to functional testing and structural testing. And [4] reported that program understanding may represent as much as 90% of the cost of maintenance. Hence, effective graphical representations can have a dramatic effect on the cost of software systems.

While there are numerous graphical notations for source code — [5] cites many notations that have been developed and used in the past — their use has seen a marked neglect by business and industry in the U.S. in favor of non-graphical PDL. A lack of automated support and the results of several studies conducted in the seventies which found no significant difference in the comprehension of algorithms represented by flowcharts and pseudo-code [6] have been major factors in this underutilization. However, automation is now available in the form of numerous CASE tools and later empirical studies reported in [7, 8] have concluded that graphical notations may indeed improve the comprehensibility and overall productivity of software. The study reported in [8] involved a well- controlled experiment in which deeply nested if-the-else constructs, represented in structured flowcharts and pseudo-code, were read by intermediate-level students. Scores for the flowchart were significantly higher than those for the pseudo-code. The statistical studies reported in [7] involved several tree-structured diagrams (e.g., PAD, YACC II, and SPD) widely used in Japan which, in combination with their environments, have led to significant gains in productivity. The results of these more recent studies suggest that the use of

0-8186-7111-4/95 $4.00

```
task body TASK_NAME is

begin
   loop
      for p in PRIORITY loop
         select

            accept REQUEST(p) (D : DATA) do

               ACTION (D);

            end;
            exit;

         else
               null;

         end select;
      end loop;
   end loop;
end TASK_NAME;
```

Figure 1: Ada task in plain source code.

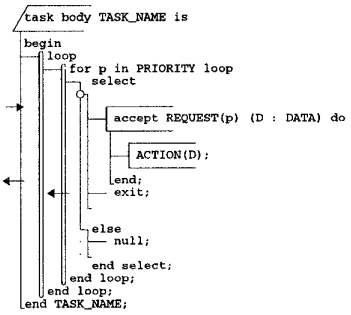

Figure 2: Task rendered as a CSD.

graphical notations with appropriate automated support should provide considerable increases in productivity over current non-graphical approaches.

A significant graphical representation that has been developed in recent years is the control structure diagram [9]. The control structure diagram is intended specifically for the graphical representation of algorithms in detailed designs as well as actual source code. The primary purpose of the control structure diagram is to reduce the time required to comprehend software by clearly depicting the control constructs and control flow at all relevant levels of abstraction. This is accomplished by using a concise set of compact, intuitive graphical constructs that appear as companions to lines of source code and do not disrupt the familiar appearance of pretty-printed source code. Initial studies [10] suggest that the control structure diagram can have a substantial positive effect on comprehension of software.

As an example of the expressive power of the control structure diagram, consider the Ada source code in Figure 1. This figure contains Ada source code adapted from [11] featuring a simple level of complexity using a concurrent control structure, the task rendezvous. The code in Figure 1 loops through a priority list attempting to accept selectively a Request with priority P. Upon acceptance, some action is taken, followed by an exit from the priority list loop to restart the loop with the first priority. In typical Ada task fashion, the priority list loop is contained in an outer infinite loop.

This short example contains two threads of control: the rendezvous, which enters and exits at the accept statement, and the thread within the task body. In addition, the priority list loop contains two exits: the normal exit at the beginning of the loop when the priority list has been exhausted, and an explicit exit invoked within the select statement. While the concurrency and multiple exits are useful in modeling the solution, they do increase the effort required of the reader to comprehend the code.

Figure 2 visualizes the source code from Figure 1 as a control structure diagram, which increases comprehensibility. This is especially useful with the multiple threads of control introduced by tasking, as well as the nested forms of control found in both concurrent and sequential programs.

3 Markup Languages

Markup languages have been used for a number of years as the basis for specifying and processing *structured documents*, that is, documents having a standard or orderly structure. The general thesis of markup is that this orderly structure can be made explicit, perhaps in the form of a context-free grammar. Markup languages allow the structure of a document to be specified by a set of identifying *tags*. Tags, each of which have an associated generic identifier, indicate the logical structure of documents by enclosing each structural element by a start and end symbol containing the appropriate generic identifier.

Although active for some time, work in markup languages has been made a popular attraction recently by internet tools based on Hypertext Markup Language (HTML) [12]. Tools such as TEX[13] have also served to increase popular awareness of the general ideas of markup languages. An excellent survey of issues involved in markup languages and structured documents can be found in [14].

An extremely important occurrence in markup language research has been the development of SGML — Standard Generalized Markup Language [15] — and its wide acceptance by government agencies and electronic publishers. SGML is an ISO standard for specifying markup languages. Many tools have been developed that are based on SGML and can process SGML-specified documents. SGML also addresses the particular needs of marking-up hypertext and multimedia documents through the HyTime [16] standard.

SGML, HyTime and related efforts have led to a great deal of research in building tools to process structured documents. One promising area of structured document research is that of using markup as a data model on which *text databases* can be built. Collections of tagged documents can be treated as a text database and queried much like conventional databases [17].

Despite the volume of research in structured documents and markup languages, the application of

```
procedure BinarySearch (Key : in KeyType; A : in ArrayType; WhereFound :
   out integer) is
   low, high, middle : integer;

begin
   WhereFound := 0;
   low := A'First;
   high := A'Last;

   while (WhereFound = 0) and (low <= high) loop
      middle := (low + high) / 2;
      if (Key < A(middle)) then
         high := middle - 1;
      elsif (Key > A(middle)) then
         low := middle + 1;
      else
         WhereFound := middle;
      end if;
   end loop;

end BinarySearch;
```

Figure 3: Sample Ada source code.

```
<_PROCBODY_>
<_PROCHEADING_>
procedure BinarySearch (Key : in KeyType; A : in ArrayType; WhereFound :
   out integer) is
</_PROCHEADING_>
<_DECLARATION_>low, high, middle : integer; </_DECLARATION_>

<_PROCSTMTPART_> begin   <_STMT_>WhereFound := 0;</_STMT_>
   <_STMT_>low := A'First;</_STMT_>
   <_STMT_>high := A'Last;</_STMT_>

   <_PRE_TEST_LOOP_>while (WhereFound = 0) and (low <= high) loop
      <_STMT_>middle := (low + high) / 2;</_STMT_>
      <_IF_>if (Key < A(middle)) then
         <_STMT_>high := middle - 1;</_STMT_>
      <_ELSE_>elsif (Key > A(middle)) then
         <_STMT_>low := middle + 1;</_STMT_></_ELSE_>
      <_ELSE_>else
         <_STMT_>WhereFound := middle;</_STMT_></_ELSE_>
      end if;</_IF_>
   end loop;</_PRE_TEST_LOOP_>

end BinarySearch;</_PROCSTMTPART_>
</_PROCBODY_>
```

Figure 4: Source code tagged using GRASP-ML.

markup languages to software engineering tools and environments is just beginning. Effective use of markup can provide important benefits to software engineering efforts.

4 GRASP-ML

GRASP-ML is a markup language designed to capture information needed to automatically produce static visualizations of program control from source code and PDL. GRASP-ML tags are automatically inserted into source code or PDL to identify all control structures. The tagged source can then be rendered as an appropriate program visualization such as the control structure diagram. The latest release of the GRASP/Ada tool employs GRASP-ML to identify all structural elements necessary to reverse engineer control structure diagrams from Ada source code.

The syntax of GRASP-ML tags essentially follows that specified by SGML. All tags have associated with them a unique identifier whose name suggests the structural element which its tag identifies plus a start and end symbol. The start symbol has the form <_identifier_> and the end symbol has the form </_identifier_>. For example, the tag that designates an if statement has "IF" as its unique identifier, <_IF_> as its start symbol, and </_IF_> as its end symbol. Each element of control has its own tag.

The source code in Figure 3 will be used to demonstrate the use of GRASP-ML. Figure 4 shows this source code tagged using GRASP-ML. The embedded tags of the markup language make all the control structures explicit and captures the control flow at a higher level of abstraction than the source code.

To automatically produce an appropriate visualization (such as the control structure diagram) of the source code, the marked-up version rather than the plain source code can be processed. Indeed, the entire process of rendering a control structure diagram can be based on the markup language rather than the source language. This GRASP-ML model of static program visualization separates graphical rendering from source language processing, and thus is language-independent [18].

In the GRASP/Ada tool, the process of reverse engineering control structure diagrams from Ada source

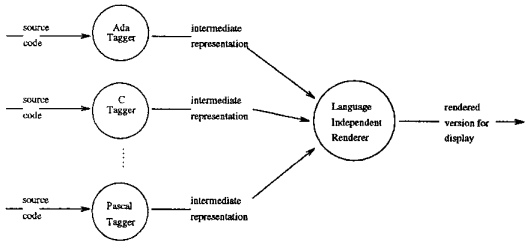

Figure 5: Language independent model.

code is based on this GRASP-ML model. Thus, control structure diagrams are reverse engineered from Ada source code in two distinct and independent phases: the *markup phase* and the *rendering phase*, as pictured in Figure 5. During the markup phase appropriate GRASP-ML tags are used to bracket all Ada control structures present in the source code. This phase is implemented by an Ada "Tagger", as depicted in Figure 5. This tagger is an Ada parser/scanner that embeds GRASP-ML tags in Ada compilation units or in fragments of Ada code.

The rendering phase essentially "interprets" the markup language. During this phase the input is scanned not for Ada constructs but for GRASP-ML tags. Each tag encountered gives information about how the control structure diagram should currently appear, and the control structure diagram is rendered to the screen as the tags are interpreted.

As an example, suppose the source code in Figure 3 were loaded as input to the GRASP/Ada tool. The Ada tagger processes this code and produces the marked-up version of Figure 4 as an intermediate step. This marked-up version is then dynamically rendered as the control structure diagram in Figure 6.

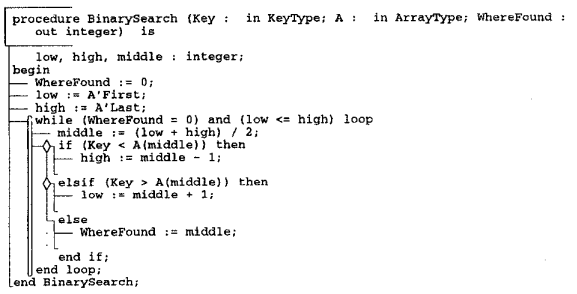

Figure 6: Source code rendered as a control structure diagram.

5 Benefits and Future Directions

Significant benefits are anticipated from this research. The development of GRASP-ML is an important contribution to both reverse engineering tools and program visualization tools. Being a meta-language, GRASP-ML can free these tools from their present language-dependence. This research also produces benefits such as defining a basis for query processing on source code and building hyperlinked webs of source code modules — all of which provide significant benefits and power to reverse engineering tools.

Reverse engineering and visualization tools based on GRASP-ML, such as GRASP/Ada Version 4, can provide all the advantages of past and current systems but in a language-independent manner. The implication of this should not be underestimated. A reverse engineering effort in which modules of the software system are written in different source languages would be aided tremendously by the availability of a single tool to provide visualization of all modules, regardless of source language. Rather than having to acquire and learn a suite of tools for the different languages needed, the software engineers involved in such an effort would only have to acquire and learn a single tool. GRASP-ML makes this possible.

Holophrasting is a technique particularly beneficial to program comprehension in which program constructs are "collapsed" or made temporarily invisible. This allows source code to be viewed at user-selected granularities where the desired level of detail is displayed in any area of the code. Through the use of GRASP-ML, holophrasting can now be provided in a language-independent manner. Tools based on GRASP-ML can collapse or expand the source code view based on language constructs such as loops, if statements, and subprogram calls without regard for what language has been used to compose the source code.

Although this research focuses on using GRASP-ML to generate control structure diagrams from source code modules, GRASP-ML could be extended to capture information needed for other types of visualizations, again in a completely language-independent manner. Two examples are architectural level diagrams and visualizations of *program plan* [19] information. GRASP-ML, or an extended form, could be used as the basis of a tool to produce architectural level diagrams such as structure charts and object diagrams. Extending GRASP-ML to capture program plan information would allow a software engineer to graphically view source code at a high degree of abstraction, yielding a significant positive effect on program comprehension and understanding.

Another benefit of this research is the possible standardization of the program visualization information represented by GRASP-ML. By specifying GRASP-ML as a SGML Document Type Definition (DTD), standard tools such as the Waterloo Database Browser [17] (or even versions of commercial wordprocessors such as WordPerfect) could be used to view the source code and its graphical representation. This allows third party document browsers that are SGML-aware to be used to view and create GRASP-ML based source code documents, thus eliminating the dependence on proprietary tools.

Not only this, but specifying GRASP-ML in SGML allows source code modules or groups of modules to be treated as a text database of program information. This would allow queries such as "*What is the return type of function F1 in module M1?*" to be automatically processed by a text database tool such as the Waterloo Database Browser.

The utility of hypertext in software engineering environments has been documented in the literature [20, 21]. HyTime [16] is a SGML-based markup language that allows definition, presentation, and browsing of electronic documents with embedded hyperlinks. Tools based on an SGML-specified GRASP-ML would be able to navigate through a automatically generated and/or user- defined web of source code modules and documentation.

6 Conclusion

GRASP-ML is a markup language used to capture all the information needed to reverse engineer graphical representations from source code. The latest release of GRASP/Ada uses GRASP-ML as the basis of its implementation to automatically generate control structure diagrams from Ada source code, thus demonstrating the feasibility of the GRASP-ML model. Basing a reverse engineering tool on the GRASP-ML yields several advantages. Among these advantages is language-independence; the generation of graphical representations is based solely on the markup tags rather than on the source language constructs. Future work will only increase the effectiveness of GRASP-ML as a tool for reverse engineering and program visualization.

References

[1] E. Chikofsky and J. Cross, "Reverse engineering and design recovery — a taxonomy," *IEEE Software*, pp. 13–17, Jan. 1990.

[2] J. H. Cross, "Reverse engineering control structure diagrams," in *Proceedings of International Workshop on Reverse Engineering*, pp. 107–116, 1993.

[3] R. Selby, "A comparison of software verification techniques," NASA Software Engineering Laboratory Series SEL-85-001, Goddard Space Flight Center, Greenbelt, Maryland, 1985.

[4] T. Standish, "An essay on software reuse," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 9, pp. 494–497, 1985.

[5] J. Martin and C. McClure, *Diagramming Techniques for Analysts and Programmers.* Prentice-Hall, 1985.

[6] B. Schneiderman, "Experimental investigations of the utility of detailed flowcharts in programming," *Communications of the ACM*, no. 20, pp. 373–381, 1977.

[7] M. Aoyama, "Design specification in japan: Tree-structured charts," *IEEE Software*, pp. 31–37, Mar. 1989.

[8] D. A. Scanlan, "Structured flowcharts outperform pseudocode: An experimental comparison," *IEEE Software*, pp. 28–36, Sept. 1989.

[9] J. Cross, S. V. Sheppard, and W. H. Carlisle, "Control structure diagrams for ada," *Journal of Pascal, Ada, and Modula 2*, vol. 9, Sept. 1990.

[10] J. Cross, "Update of grasp/ada reverse engineering tools for ada," tech. rep., Auburn University, Dec. 1993.

[11] J. G. P. Barnes, *Programming in Ada.* Menlo Park, CA: Addison-Wesley, 2 ed., 1984.

[12] T. Berners-Lee, "Hypertext markup language." Internet Draft, July 1993.

[13] D. E. Knuth, *The TEXBook.* Addison-Wesley, 1986.

[14] J. André, R. Furuta, and V. Quint, eds., *Structured Documents.* Cambridge University Press, 1989.

[15] C. Goldfarb, *The SGML Handbook.* 1990.

[16] S. R. Newcomb, N. A. Kipp, and V. T. Newcomb, "The hytime hypermedia/time-based document structuring language," *Communications of the ACM*, vol. 34, pp. 67–83, Nov. 1991.

[17] E. W. Mackie, "Waterloo text database, system overview," Tech. Rep. 94–12, University of Waterloo, Computer Science Department, Waterloo, Ontario, Mar. 1994.

[18] D. Hendrix and J. Cross, "Language independent generation of graphical representations of source code," in *Proceedings of the 23rd Annual ACM Computer Science Conference*, 1995 (to appear).

[19] L. Wills, "Automated program recognition: A feasibility demonstration," *Artificial Intelligence*, vol. 45, pp. 113–68, Sept. 1990.

[20] J. Bigelow, "Hypertext and case," *IEEE Software*, pp. 23–27, Mar. 1988.

[21] J. L. Cybulski and K. Reed, "A hypertext based software engineering environment," *IEEE Software*, Mar. 1992.