Visual Support for Incremental Abstraction and Refinement in Ada 95

T. Dean Hendrix, James H. Cross II, Larry A. Barowski, and Karl S. Mathias
Computer Science and Engineering
107 Dunstan Hall
Auburn University, AL 36849
{hendrix, cross, larrybar, mathiks}@eng.auburn.edu

1. ABSTRACT

GRASP is a software engineering tool which uniquely combines a source code diagramming technique, the control structure diagram (CSD), with other comprehension aids such as complexity visualization, syntax coloring and source code folding. The synergistic combination of these features in GRASP has the potential to be a powerful aid in any activity where source code is expected to be read. The primary focus of GRASP is to improve the comprehension efficiency of software and, as a result, improve reliability and reduce costs during design, implementation, testing, maintenance and reengineering.

1.1 Keywords

Software visualization, folding, program understanding

2. INTRODUCTION

Software visualization is an active area of research that investigates efficient and effective ways of automatically producing graphical representations of program source code, algorithms, or the runtime behavior of software. Such visualization technology promises to have a positive impact on difficult and costly issues in software engineering such as communicating program information for the purposes of testing, maintenance, and reengineering [4, 8]. Indeed, the benefit of using graphical representations of software as comprehension aids has long been acknowledged, especially in the context of large, complex systems.

Visualization in and of itself, however, is not necessarily beneficial [19]. There are many issues that influence the utility of software visualization. Some are practical and cognitive issues relating to the user of the visualization and the process of human comprehension [2, 6, 22]. Other issues include those which relate to the nature of a particular visualization itself [9, 20].

Building on prior successful results of the GRASP (graphical representations of algorithms, structures, and processes) research project, a current focus of GRASP is on providing an effective combination of source code visualization and source code folding. The roles of such comprehension aids in design, implementation, testing, maintenance and reengineering are of particular interest. Specifically, those activities in which program understanding plays a significant role are expected to benefit through the effective use of GRASP's visualization and folding.

3. VISUALIZING CONTROL STRUCTURE

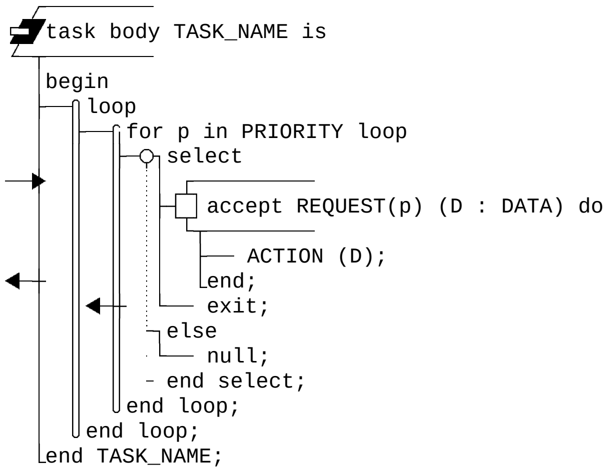
The CSD is the principal visualization in the GRASP environment. GRASP provides automatic generation of CSDs from Ada 95, C, C++, Java, and VHDL source code. A major objective in the philosophy that guided the development of the CSD was that the graphical constructs should supplement the source code without disrupting its familiar appearance. That is, the CSD should appear to be a natural extension of the source code and, similarly, the source code should appear to be a natural extension of the diagram. This has resulted in a concise, compact graphical notation which attempts to combine the best features of diagramming with those of well-indented source code.

The CSD has the potential to replace traditional pretty-printed source code in soft and hard copy documents. In an initial evaluation [9], the CSD was compared to four other graphical representations for algorithms (ANSI flowchart, Nassi-Shneiderman Diagram, Warnier-Orr Diagram, and Action Diagram) against eleven performance characteristics. The CSD was clearly preferred to all other graphical representations in a majority of the performance characteristics.

A comparison of the CSD with plain text source code is shown in Figure 1 and Figure 2. Figure 1 contains Ada 95 source code adapted from [5]. Figure 2 contains that same source code rendered as a CSD. While the same structural and control information is available in both figures, the CSD makes the control structures and control flow more visually apparent than does the plain text alone, and it does so without disrupting the conventional layout of the source code.

```
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 95 source code



```
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 2. Ada 95 source code rendered as a CSD

The power of the CSD is much more evident in larger and/or more complex source code. For example in large programs, especially those which are a part of legacy systems, it is not uncommon for complex control structures such as loops to span hundreds of lines. The physical separation of sequential components within these large control structures becomes a significant obstacle to comprehension. The CSD clearly delineates each control structure and provides context and continuity for the sequential components nested inside, thus increasing comprehension efficiency. With additional levels of nesting and increased physical separation of sequential components, the visibility of control constructs and control paths becomes increasingly obscure, and the effort required of the reader dramatically increases in the absence of the CSD.

4. SUPPORTING INCREMENTAL ABSTRACTION

Many researchers have focused considerable effort on discovering and describing the process by which software professionals comprehend source code and other software artifacts. Although many open questions remain and there is still much uncertainty regarding various proposed comprehension models, there do exist common and well-accepted elements of comprehension processes.

One such commonly accepted element of program comprehension is what we will refer to as *incremental abstraction*, that is, the process of successively building up mental representations of various levels of abstractions of source code text structures and their relationships. These mental representations are often called *chunks* have been shown in the literature to play significant roles in human comprehension of software [3,7,10,11,15,16,18,21].

The study of fundamental cognitive processes has revealed well-defined limits on the capacity of the human mind during comprehension tasks. Miller [16] described the classic 7±2 limit on short-term memory and others have demonstrated that this capacity diminishes as task complexity increases [15]. Thus, grouping portions of source code together into chunks allows the chunk of code to be understood abstractly as a unit rather than through the complete details of its components. This concept of chunking builds upon itself in that lower-level chunks are combined to form higher-level chunks, thus providing a method of comprehending a program in terms of various levels of abstraction in a systematic way, while addressing the inherent limits on the capacity of the human mind.

Before chunks can be used as part of a comprehension strategy, they must be identified. The process of scanning through the source code, in either a forward or backward direction, to identify appropriate chunks is known as tracing [7]. Although chunks may not be delineated by particular syntax boundaries in the source code, they are often delineated by control structures and program modules. The CSD directly supports the tracing process by providing clear and intuitive visual cues to both control structure boundaries and program module boundaries. This visual support can be especially significant when tracing large regions of code where single control structures might span hundreds of lines of code.

The identification and comprehension of chunks involves both semantic and syntactic knowledge [21]. Semantic knowledge is relatively independent of any particular language and involves understanding basic concepts such as looping structures and fundamental algorithms, and recognizing design patterns in code. Syntactic knowledge is language specific and allows semantic structures to be recognized in a particular language [7]. A clear relationship exists between the syntactic form of source code and the ability of human readers to construct useful mental abstractions from that source code [13]. Source code that is well structured and visually appealing facilitates the tracing and chunking process. The intuitive visualization of the CSD, displayed as a companion to well-indented, pretty-printed source code thus provides direct support for chunking and tracing during program comprehension.

The visual presentation of a chunk can have a significant impact on its understandability [14]. Chunks that are visually distinct from the rest of the program are much easier to comprehend. GRASP provides direct support of this concept by allowing the CSD to be viewed in user-selectable granularities. A user can select portions of code according to control structure boundaries, program module boundaries, or arbitrary boundaries, and then fold them into the single CSD symbol shown in Figure 3.

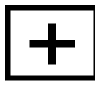


Figure 3. CSD folding symbol

Thus, a user of GRASP can fold the CSD into levels of abstractions, or chunks, as the source code is being read and understood. The folded portions of source code can then be labeled with a comment or with the source language construct that begins the chunk (e.g., a procedure heading).

Figure 4 illustrates folded CSDs for a procedure body, a block, an if-then, and a while loop. The folded portions of the construct can be arbitrarily large and contain regions of code that are themselves folded. Thus, folding in GRASP is implemented as a natural form of outlining in which the code is folded and unfolded based on the control structures in the CSD. What makes this approach so effective is the visual clarification of the chunks resulting from the folded CSD. That is, the folds are visually labeled by CSD symbols, much as chunks are abstractly labeled according to functionality.

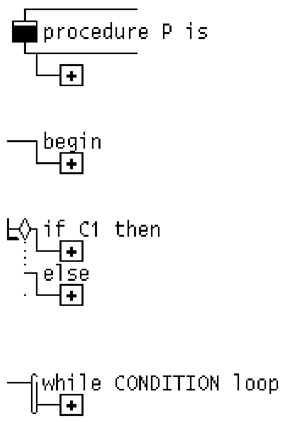
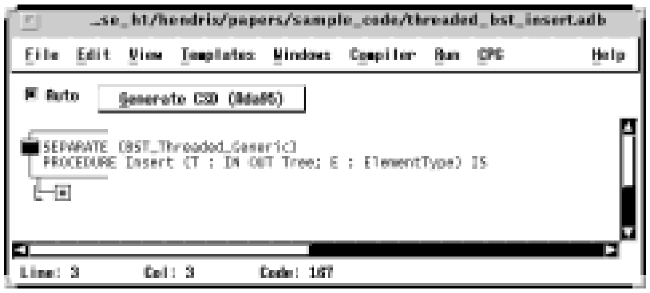


Figure 4. Folded CSDs

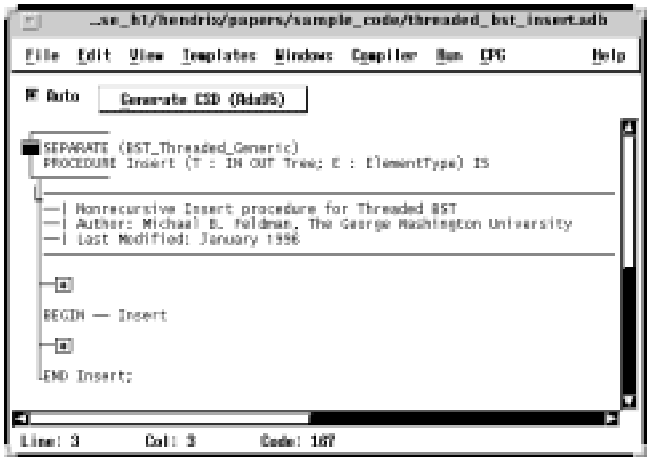
As an example of the use of folding, Figures 5 through 8 show the presentation of Ada 95 source code in increasing levels of detail, as the code might be presented in a classroom setting. The source code is taken from the text by Michael Feldman [12] used in the CS 2 (data structures) course at Auburn University. Folding offers a pedagogically sound technique for teaching algorithms presented in source code. GRASP provides students with a robust and efficient tool to facilitate reading code and learning algorithms through natural chunking processes.

Figure 5 shows a GRASP CSD window displaying the source code for inserting a new element into a threaded binary search tree. The source code has been completely folded except for the procedure heading. This hides all the detail of the algorithm by abstracting away everything except the public interface to the implementation. Figure 6 unfolds the CSD to the next level of detail where the module summary comments are shown but where both the declarations and the executable statement section are folded. Figure 7 shows the unfolded declaration section, revealing a nested subprogram. Figure 8 shows the declaration section again folded, but the statement section expanded to reveal two subchunks, each identified by the CSD folding symbol and a leading comment.



```
SEPARATE (BST_Threaded_Generic)
PROCEDURE Insert (T : IN OUT Tree; E : ElementType) IS
```

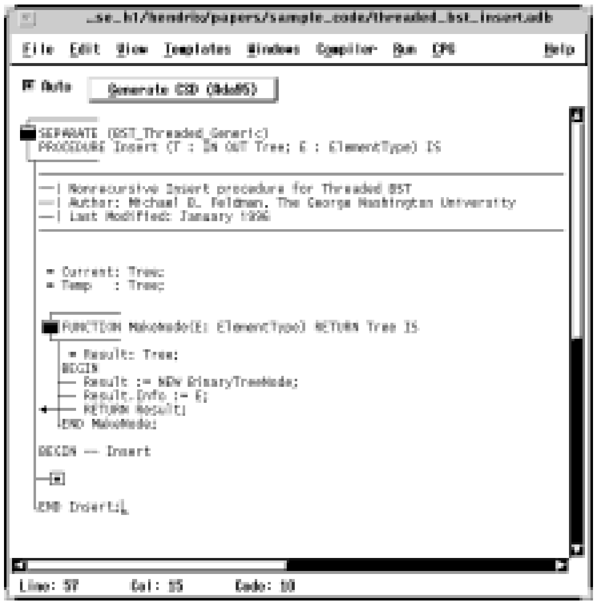
Figure 5. Source code folded to module heading


..se_h1/hendrix/papers/sample_code/threaded_bst_insert.adb
File Edit View Templates Windows Compiler Run CPG Help
Auto
Generate CSD (Ada95)

```
SEPARATE (BST_Threaded_Generic)
PROCEDURE Insert (T : IN OUT Tree; E : ElementType) IS

--| Nonrecursive Insert procedure for Threaded BST
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996

BEGIN -- Insert

END Insert;
```

```
Line: 3        Col: 3        Code: 167
```

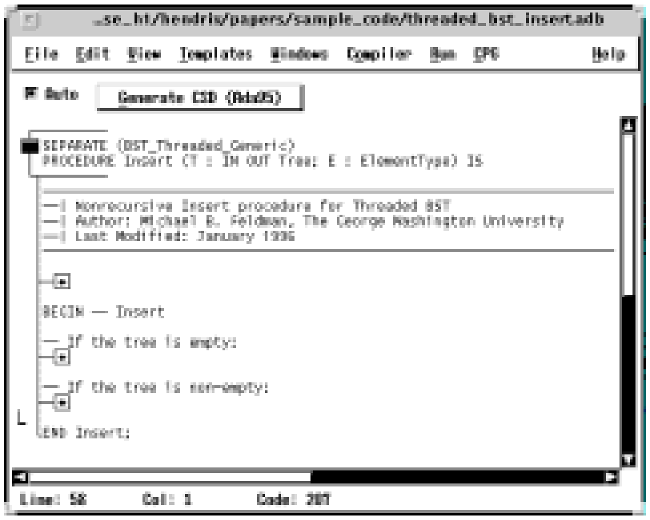
Figure 6


..se_h1/hendrix/papers/sample_code/threaded_bst_insert.adb
File Edit View Templates Windows Compiler Run CPG Help
Auto
Generate CSD (Ada95)

```
SEPARATE (BST_Threaded_Generic)
PROCEDURE Insert (T : IN OUT Tree; E : ElementType) IS

--| Nonrecursive Insert procedure for Threaded BST
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996

 Current: Tree;
 Temp   : Tree;

FUNCTION MakeNode(E: ElementType) RETURN Tree IS
  Result: Tree;
 BEGIN
   Result := NEW BinaryTreeNode;
   Result.Info := E;
   RETURN Result;
 END MakeNode;

BEGIN -- Insert

END Insert;
```

```
Line: 57       Col: 15       Code: 10
```

Figure 7


..se_h1/hendrix/papers/sample_code/threaded_bst_insert.adb
File Edit View Templates Windows Compiler Run CPG Help
Auto
Generate CSD (Ada95)

```
SEPARATE (BST_Threaded_Generic)
PROCEDURE Insert (T : IN OUT Tree; E : ElementType) IS

--| Nonrecursive Insert procedure for Threaded BST
--| Author: Michael B. Feldman, The George Washington University
--| Last Modified: January 1996

BEGIN -- Insert

-- If the tree is empty:

-- If the tree is non-empty:

END Insert;
```

```
Line: 58       Col: 1        Code: 207
```

Figure 8

Applications in classroom pedagogy not withstanding, folding is expected to be particularly useful as a form of stepwise abstraction during code reviews and other such activities in an industrial setting. The automated support for folding provided by GRASP also has direct application to formal software development methods [1] and Cleanroom correctness verification [17].

5. SUPPORTING INCREMENTAL DEVELOPMENT

A goal of the GRASP research project has been to tightly integrate support for both forward engineering and reverse engineering in a single automated environment. This integration is clearly demonstrated in the combination of diagramming and folding. The same features of GRASP which support incremental abstraction during comprehension tasks also support incremental development during forward engineering processes.

Indeed, the example of incremental abstraction presented in Figures 5 through 8 could easily be transformed into an example of stepwise refinement. Users of GRASP have direct visual support for incremental development and stepwise refinement through folded CSDs. Users can create initially folded CSDs to represent regions of code that need refinement or elaboration. As these regions are incrementally refined, they can be individually folded again to reduce the visual clutter and allow the user to focus on the current region of code being developed.

6. FUTURE WORK

An overriding goal of the GRASP project is to produce tools and results which are useful in production settings with large software projects. A major obstacle to providing effective visualization for large software systems is the scalability of the visualizations themselves. Many visualization techniques such as dynamic algorithm animation do not scale up to large systems in an effective way.

Static code visualizations which can be automatically generated very efficiently, such as the CSD, show promise for addressing the visualization of large systems. A major issue for these static visualizations is providing context to the user. Often when viewing a detailed visualization of source code in a large system, the "big picture" of the software is lost. To address this issue, a context view is being developed as a companion to the CSD.

The context view provides a highly compressed view of the CSD and/or the source code (as a user-selectable combination) in a companion window. The normal CSD window is synchronized with the context view so that the portion of the source code currently displayed in detail in the CSD window is highlighted in the context view. Thus, while reading and comprehending a detailed portion of source code in a CSD window, the user is immediately able to see an overview or “landscape” of the entire program as well as where the code under examination fits in. The intent of the context view is to further increase the comprehension efficiency afforded by the CSD to large software systems. It is anticipated that the context view will be especially useful as a companion to folding.

7. CONCLUSION

The emphasis of the GRASP research project is on improving the comprehensibility of software through automatic generation of appropriate visualizations, such as the CSD, along with support for other comprehension aids such as folding. GRASP’s unique combination of these features along with emerging features such as the context view, promises to be a highly effective aid to program comprehension.

If a graphical representation effectively supports well-defined cognitive processes employed during comprehension tasks, comprehension efficiency can be increased. GRASP directly supports program comprehension through the use of chunks by enabling the CSD to be viewed at user-selectable granularities. Language constructs ranging from a single control structure such as a simple loop to the entire program unit itself can be folded or collapsed to a single CSD symbol, increasing the level of abstraction at which the code is viewed. In addition to directly supporting a user’s mental representation of a program in terms of chunks, this also allows the user to choose the amount of source code that is present at any one time in the diagram, thus increasing comprehension efficiency.

The current release of GRASP is being used extensively at Auburn University in a variety of computer science courses for software in Ada 95, C, C++, Java, and VHDL. In addition, GRASP is now freely available for download from the Internet for Solaris, SunOS, IRIX, Linux, AIX, and Windows 95/NT. GRASP has been in tremendous demand, with downloads from industry, academia, government agencies, and the general public averaging approximately 1,000 per month since August, 1997. GRASP, together with online documentation, is available at http://www.eng.auburn.edu/grasp.

8. ACKNOWLEDGMENTS

The GRASP project is funded, in part, by a grant from the National Science Foundation.

9. REFERENCES

[1] Andrews, D.J. and Ince, D.C. (1995). Transformational Data Refinement and VDM. *Information and Software Technology*, 37 (11), pp. 637-651.

[2] Arunachalam, V. and Sasso, W. (1996). Cognitive Processes in Program Comprehension: An Empirical Analysis in the Context of Software Engineering. *Journal of Systems and Software*, 34, pp. 177-189.

[3] Badre, A. (1982). Designing Chunks for Sequentially Displayed Information, in Badre, A. and Shneiderman, B. (eds.), *Directions in Human Computer Interaction*, Ablex Publishing, pp. 179-193.

[4] Baecker, R. M., Digiano, C., and Marcus, A. (1997). Software Visualization for Debugging. *Communications Of The ACM*, 40, 4, pp. 44-54.

[5] Barnes, J. G. P. (1984) *Programming in Ada, Second Edition*, Menlo Park, CA: Addison-Wesley.

[6] Basili, V., and Selby, R. (1987). Comparing the Effectiveness of Software Testing Strategies. *IEEE Transactions on Software Engineering*, SE-13, 12, pp.1278-1296.

[7] Cant, S.N., Jeffery, D.R., and Henderson-Sellers, B. (1995). A Conceptual Model of Cognitive Complexity of Elements of the Programming Process. *Information and Software Technology*, 37 (7), pp. 351-362.

[8] Cross, J. H. (1993). Improving Comprehensibility of Ada With Control Structure Diagrams. *Proceedings of Software Technology Conference*, April 11-14, 1994, Salt Lake City, UT (distributed on CD-ROM) 25 pages.

[9] Cross, J.H., Maghsoodloo, S., and Hendrix, T.D. (1998). The Control Structure Diagram: An Initial Evaluation. Empirical Software Engineering. (to appear).

[10] Davis, J.S. (1984). Chunks: A Basis for Complexity Measurement. *Information Processing and Management*, 20 (1), pp. 119-127.

[11] Ehrlich, K. and Soloway, E. (1984). An Empirical Investigation of the Tacit Plan Knowledge in Programming. in Thomas, J.C. and Schneider, M.L. (eds.), *Human Factors in Computer Systems*, Ablex Publishing, pp. 113-133.

[12] Feldman, M.B. (1997). Software Construction and Data Structures with Ada 95. Addison-Wesley.

[13] Gilmore, D.J. and Green, T.R.G. (1985). The Comprehensibility of Programming Notations. in Shackel, B. (ed.), Human-Computer Interaction - Interact ’84 IFIP, pp. 461-464.

[14] Green, T.R.G, Sime, M.E., and Fitter, M.J. (1981). The Art of Notation. in Coombs, M.J. and Alty, J.L. (eds.), Computing Skills and the User Interface, Academic Press, pp. 221-251.

[15] Kintsch, W. (1977). *Memory and Cognition*, John Wiley.

[16] Miller, G.A. (1956). The Magic Seven Plus or Minus Two. Some Limits on Our Capacity for Processing Information. *Psychological Review*, 63, pp. 81-97.

[17] Mills, H.D. (1988). Stepwise Refinement and Verification in Box-Structured Systems. IEEE Computer, June, pp. 23-36.

[18] Pennington, N. (1987). Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. *Cognitive Psychology*, 19, pp. 295-341.

[19] Petre, Marian (1995). Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38, 6, pp. 33-44.

[20] Raymond, D. (1991). Characterizing Visual Languages. *Proceedings of the 1991 IEEE Workshop on Visual Languages*, pp. 176-182.

[21] Shneiderman, B. and Mayer, R. (1979). Syntactic/Semantic Interactions in Programmer Behavior: A Model and Experimental Results. Int. Journal of Computer and Information Sciences, 8 (3), pp. 219-238.

[22] von Mayrhauser, A., and Vans, A.M. (1996). Identification of Dynamic Comprehension Processes During Large Scale Maintenance. *IEEE Transactions on Software Engineering*, 22, 6, pp. 424-437.