Visualization and Measurement of Source Code

James H. Cross II, Kai H. Chang, T. Dean Hendrix, and Richard O. Chapman
Auburn University
Patricia A. McQuaid
California Polytechnic State University at San Luis Obispo

The GRASP (Graphical Representations of Algorithms, Structures, and Processes) project, which has successfully prototyped a new algorithmic-level graphical representation for software—the control structure diagram (CSD)—is currently focused on the generation of a new fine-grained complexity metric called the complexity profile graph (CPG). The primary impetus for creation and refinement of the CSD and the CPG is to improve the comprehension efficiency of software and, as a result, improve reliability and reduce costs. The current GRASP release provides automatic CSD generation for Ada 95, C, C++, Java, and Very High-Speed Integrated Circuit Hardware Description Language (VHDL) source code, and CPG generation for Ada 95 source code. The examples and discussion in this article are based on using GRASP with Ada 95.

Computer professionals have long promoted the idea that graphical representations of software can be extremely useful as comprehension aids when used to supplement textual descriptions and specifications of software, especially for large complex systems. The general goal of the GRASP research project is the investigation, formulation, and generation of graphical representations of algorithms, structures, and processes for source code written in languages such as Ada 95, C, C++, Java, and VHDL.

The CSD and the CPG

This article focuses on the generation or reverse engineering of CSDs and CPGs from source code for visualization and measurement. The CSD is an algorithmic-level graphical representation for software. The CPG is a new visualization of a fine-grained complexity metric. By synchronizing the CSD and the CPG, the CSD view of control structure, nesting, and source code is directly linked to the corresponding visualization of statement-level complexity in the CPG.

The CSD has been designed to be an intuitive, compact, and nondisruptive visualization of control structure [1, 2]. The basic graphic symbols of the CSD are intuitive and suggest their purpose by their appearance. For example, diamond-shaped symbols are used to mark decisions as in the ubiquitous flowchart, and Grady Booch symbology [3] is used to mark program units such as subprograms, modules, and classes. The CSD is compact, taking up no more room on the printed page or display screen than normal, plain-text source code. The CSD is nondisruptive to the source code—the CSD retains the appearance of traditional pretty-printed source code by acting as a companion to, rather than a replacement for, the code.

The CPG is based on a set of functions that describe the context, content, and the scaling for complexity on a statement-by-statement basis [4]. When rendered graphically, the result is a composite profile of complexity for the program unit. Ongoing research includes the development and refinement of the associated functions and the development of the CPG generator prototype.

GRASP and CASE Tool Integration

The GRASP tool offers a level of flexibility suitable for experimentation, evaluation, and practical application. It is expected that GRASP will be integrated with existing computer-aided software engineering (CASE) tools in which the primary motivation for the generation of graphical representations is increased support for software lifecycle activities that range from design to maintenance with emphasis on visual verification and measurement. These activities should be greatly facilitated by an automatically generated set of formalized diagrams and graphs to supplement the source code and other forms of existing documentation.

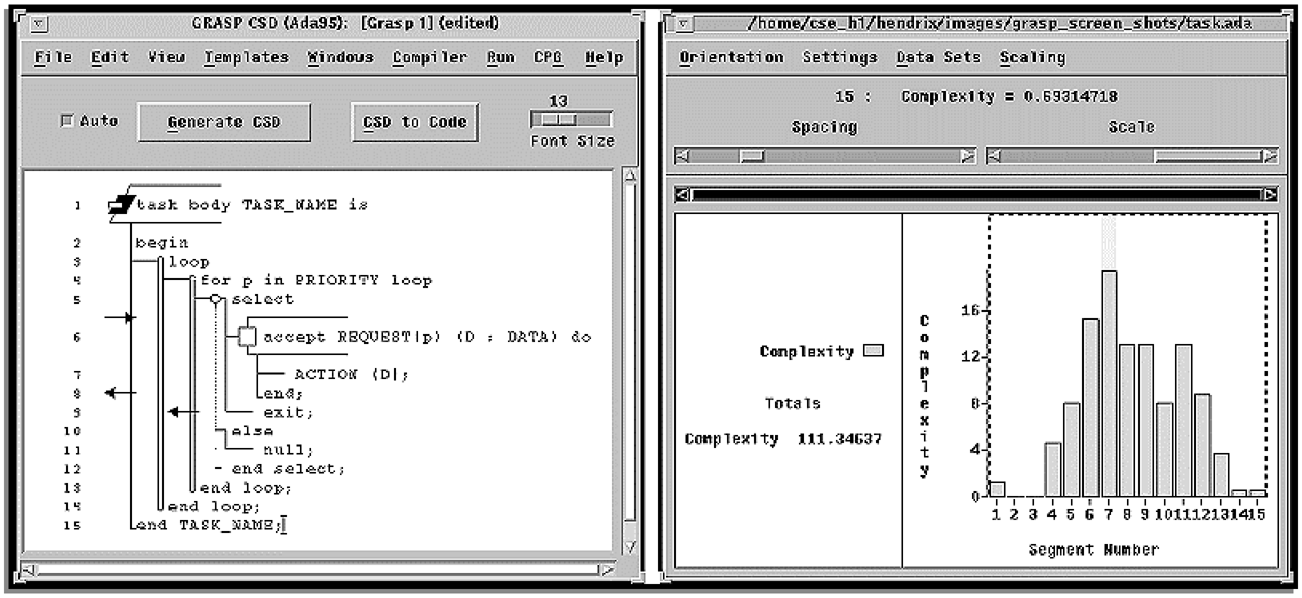
An important goal of the GRASP project is to provide the foundation for a CASE environment in which reverse engineering and forward engineering (development) are tightly coupled. In such an environment, the user may specify the software in a graphically oriented language, then automatically generate the corresponding source code. Alternatively, the user may design or review source code, then automatically generate the graphical representations either dynamically as the code is entered or as a form of post-processing. The GRASP software tool has the potential to be a powerful aid in environments where source code is expected to be written or read.

The tool is particularly suitable for activities during detailed design, implementation, testing, maintenance, and reengineering. The CSD is expected to be a valuable aid in comprehension and analysis of overall program structure and flow of control, while the CPG is expected to provide additional valuable insight by visualizing the complexity of both context and content of program elements.

Visualizing Structure and Complexity

GRASP is a continually evolving software engineering tool. The emphasis to this point has been on visualizing program structure via the automatic generation of CSDs from source code to support development, maintenance, reverse engineering, and reengineering. GRASP now provides the capability for the user to generate CSDs from Ada 95, C, C++, Java, or VHDL source code in a reverse engineering mode, as well as forward engineering mode, with a level of robustness suitable for use in a commercial environment. The use of software visualizations and graphical representations such as the CSD as aids in program comprehension tasks is well documented [5,6,7,8].

Although the CSD is useful to intuitively visualize the control structures and control paths present in source code, to fully aid program comprehension tasks, additional information needs to be visualized. It is important that the reader locate complex portions of code so that those areas may be given a more careful examination. For this reason, GRASP has been extended to automatically generate the CPG visualization of complexity.

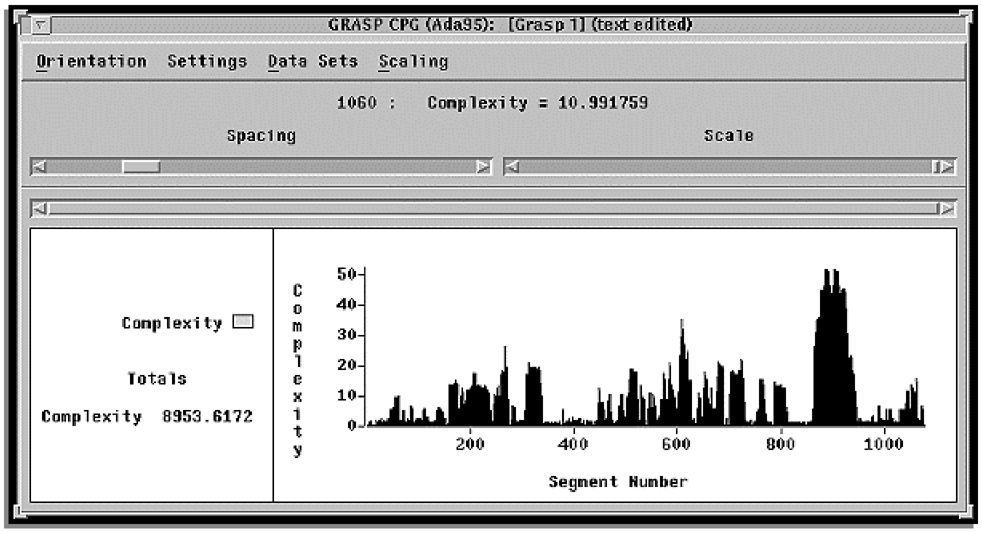

GRASP CSD (Ada95): [Grasp 1] (edited)
File Edit View Templates Windows Compiler Run CPG Help
Auto
Generate CSD
CSD to Code
13
Font Size
/home/cse_h1/hendrix/images/grasp_screen_shots/task.ada
Orientation Settings Data Sets Scaling
15 : Complexity = 0.69314718
Spacing
Scale
Complexity
Totals
Complexity 111.34637
Complexity
0
4
8
12
16
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
Segment Number

```
1   task body TASK_NAME is
2   begin
3     loop
4       for p in PRIORITY loop
5         select
6           accept REQUEST(p) (D : DATA) do
7             ACTION (D);
8           end;
9           exit;
10        else
11          null;
12        end select;
13      end loop;
14    end loop;
15  end TASK_NAME;
```

Figure 1. *Synchronized CSD and CPG in GRASP.*

Figure 1 shows a coherent and synchronized GRASP visualization of both program structure and program complexity. In the first window, an Ada tasking example from [9] is rendered as a CSD, and in the second window it is rendered as a CPG. Both windows are synchronized with each other, which allows the user to scroll one and the other will automatically scroll, or the user may select a particular location in one window and the other window will automatically highlight its corresponding location. The CPG clearly suggests that the most complex area of the code is centered at line seven, where the procedure Action is called during the rendezvous.

Each bar in the CPG represents the complexity of the corresponding source code construct displayed in the CSD window. Since the CSD and CPG are synchronized, the user can easily identify complex regions of code via the CPG and quickly navigate to them in the CSD window. This capability is especially useful in large software systems where it is difficult to measure, evaluate, and comprehend the source code as a whole. Figure 2 shows a portion of the CPG for a software system written in Ada 95 that contains approximately 3,700 lines of code. Though it cannot be considered a large system, it serves well to illustrate the CPG's potential usefulness in larger systems. The most complex region of code in this example is easily identified in the right region of the CPG where the graph "spikes." With a single click, the user can immediately access the source code for that region in a synchronized CSD window. When the user points and clicks within this complex region of the CPG, the CSD automatically scrolls to this location in the source code.


GRASP CPG (Ada95): [Grasp 1] (text edited)
Orientation Settings Data Sets Scaling
Spacing
Scale
Complexity
Totals
Complexity
0
10
20
30
40
50
200
400
600
800
1000
Segment Number

1060 : Complexity = 10.991759

Totals
Complexity 8953.6172

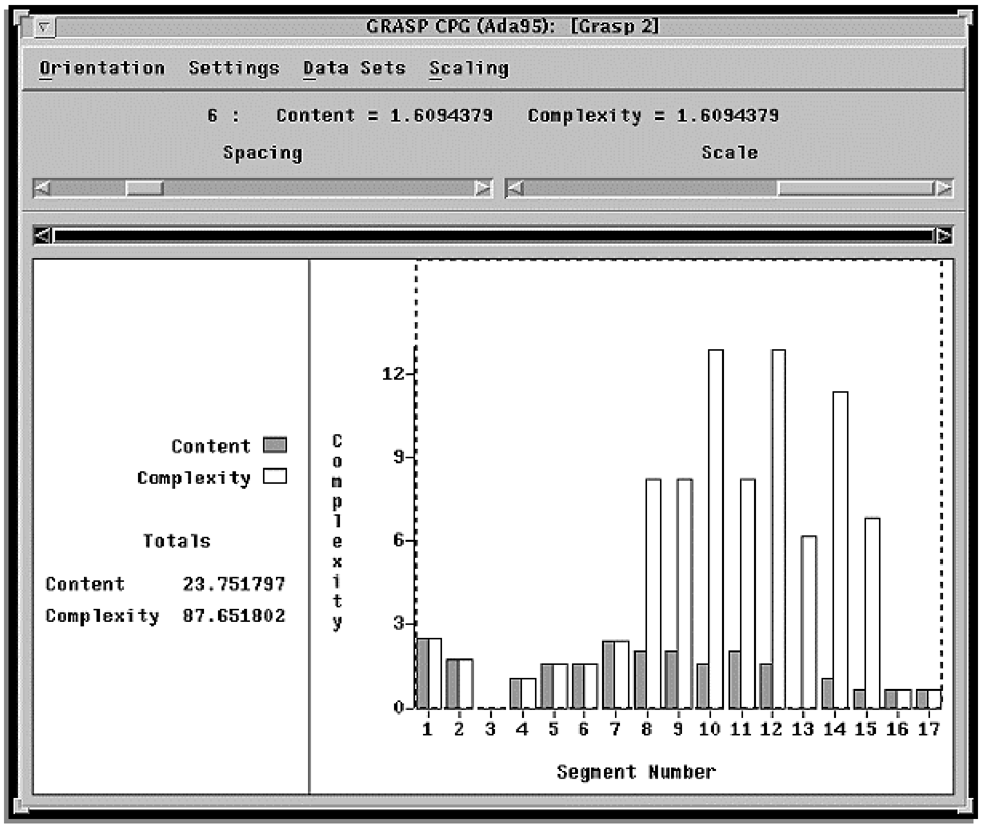
Figure 2. *Complexity Profile Graph for a larger program.*

Since the CSD window provides full-featured text editing (as well as compilation, linking, and execution of programs), the user can restructure complex areas of code, and the corresponding effect will be immediately visualized in the CPG.

Measurable Units of Software

The CPG is based on a profile metric that is designed to compute complexity at various levels of granularity based on the underlying source language. We will call these various levels of granularity *measurable units* of software. The fundamental idea of the profile concept is that software can be partitioned into a set of measurable units in such a way that each token belongs to exactly one such unit. For example, an Ada 95 program is grammatically partitioned into individual program units (package, subprogram, task, etc.), and each of these can be further partitioned into statements, etc. Theoretically, complexity can be calculated for any level of granularity defined by the grammar of the source language. In our present research, we calculate complexity at the production level in the source language grammar.

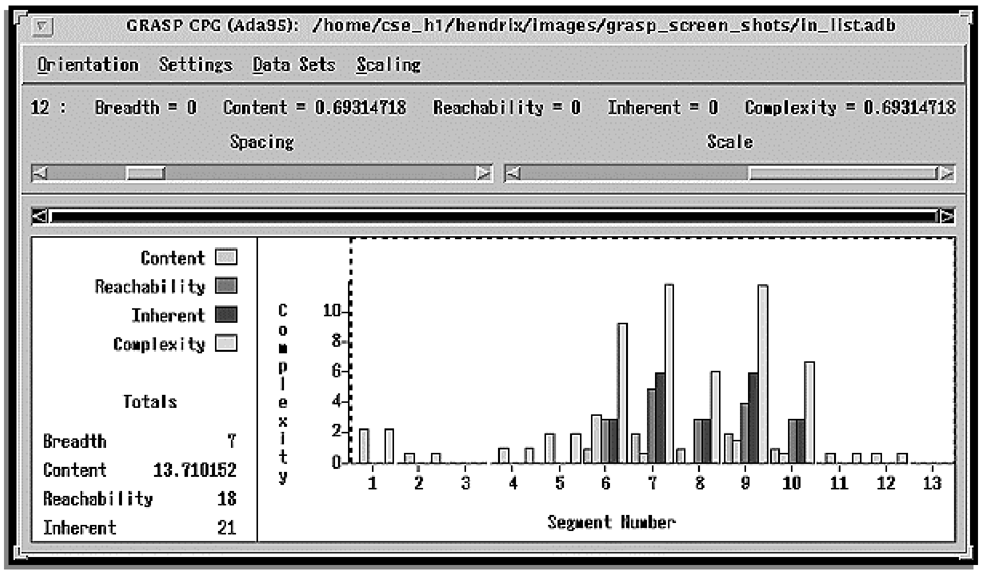
The complexity of both the *content* and the *context*
complexity measures: content complexity, inherent complexity, reachability complexity, and breadth complexity. Each of these four measures, as well as total complexity, may be plotted as a CPG independently of the others as shown in Figure 3 and Figure 4. The CPG measures are color coded when displayed on a color computer screen.



6 : Content = 1.6094379 Complexity = 1.6094379

Totals	
Content	23.751797
Complexity	87.651802

Figure 3. *CPG showing content and total complexity plotted separately.*



12 : Breadth = 0 Content = 0.69314718 Reachability = 0 Inherent = 0 Complexity = 0.69314718

Totals	
Breadth	7
Content	13.710152
Reachability	18
Inherent	21

Figure 4. *CPG showing all four measures plotted separately along with total complexity.*

Tool Verification

Visualization and measurement tools such as GRASP are nontrivial to develop. There are many, often subtle, details for which the tool could produce incorrect results. Therefore, it is important that tools such as GRASP be verified as being robust enough for practical application. GRASP includes a self-test feature that runs in batch mode (without the graphical user interface) and processes specified directories of source code. During this self-test, each component of GRASP is tested, including the lexer/parser, CSD generator, and CPG generator. Each file in the specified directory is parsed, and both the CSD and CPG are generated and checked for errors. For example, the CSD is checked for validity against a set of approximately 300 rules that describe a well-formed diagram. In addition to the various checks against the CSD and CPG, a byte-level scan of the file is performed to ensure that GRASP has not changed in any way the underlying source code.

The Ada Compiler Validation Capability (ACVC) suite is used for the Ada 95 self-test. The ACVC consists of positive tests (correct code) and negative tests (code with syntactic and semantic errors). GRASP has successfully passed the self-test on all positive and negative ACVC test files. This rigorous verification process is important to ensure a continued level of robustness and efficiency from GRASP.

Additional Capabilities

The GRASP research project continues to explore new areas of software visualization and measurement. Unit symbols based on the widely used Booch architectural diagram notation [3] have been integrated into the CSD. These architectural-level symbols at the source-code level will provide a visual link to the architecture level of the software under consideration. Part of our future research will be to automatically generate appropriate architecture-level diagrams and hyperlink them to the source code via these new CSD symbols.

We also are investigating ways that GRASP can address the problems of developing and maintaining multilingual software. A tool such as GRASP that can provide visualization and measurement for source code written in Ada, C, C++, and Java would be well suited to develop or maintain software systems with component modules that are written in more than one language. Multilingual support for CPG generation is expected to be added.

To extend GRASP to a different domain, we recently added support for VHDL. Originally developed as a hardware simulation language, VHDL has become an industry standard for specification of digital hardware as well [10]. Our implementation provides VHDL developers (or circuit designers working from VHDL code as a specification) with a graphical representation of behavioral VHDL code, similar to that provided to an Ada programmer by GRASP. Such a representation is particularly useful for the VHDL community because of the nature of the hardware design process. VHDL code is not an implementation, as in the Ada case, which is useful for its own sake, but is a specification to be read and understood by designers who must translate it into circuitry. Given this, code readability is of paramount importance for users of VHDL.

Conclusion

The emphasis of the GRASP project is automatic generation of the CSD and CPG from source code to support software lifecycle activities. These lifecycle activities should be greatly facilitated by an automatically generated set of formalized diagrams and charts to supplement the source code and other forms of documentation. Code reading is still a popular and viable verification and testing strategy as evidenced by current literature [11,12,13]. Hence, improved comprehension efficiency that results from the integration of graphical notations and source code could have a significant impact on the overall cost of software production.

The current release of GRASP provides users the capability to generate CSDs and CPGs from Ada source code with a level of flexibility suitable for practical application. The CPG has the potential to provide more useful information than traditional metrics by incorporating both the content and the context complexities into the metric. The CPG seeks to identify not only complex statements but also complex sets of statements and regions of code called *clusters*. Once the clusters are identified, paths to reach the clusters, as well as other high-complexity paths, can be identified. The primary theme of all applications is to locate and prioritize clusters for selective consideration and to concentrate efforts on denser regions where exhaustive review is impractical. This information has direct application as a form of continuous feedback for analysis to software design, implementation, testing, maintenance, and the software development process. A major thrust of our ongoing research is the empirical validation of the CPG and the underlying complexity functions for context and content. This research will help determine the overall usefulness of the CPG as well as the best activities for its application.

The GRASP software tool has been verified through a rigorous testing process using the ACVC suite. A robust tool, such as GRASP, is essential for the evaluation of the CSD and CPG on any non-trivial Ada 95 software. GRASP is being used extensively in computer science and engineering courses at Auburn University, and it has been downloaded from our Web site over 2,500 times by a diverse set of users. Version 6.2 is freely available from http://www.eng.auburn.edu/grasp
Linux, Windows95, and WindowsNT. Currently, CPG generation is provided only in GRASP for Solaris, SunOS, IRIX, and Linux.

Acknowledgements

Research on the GRASP project has been supported, in part, by grants from NASA, Defense Information Systems Agency (DISA), and Advanced Research Projects Agency (ARPA). Current graduate students working on the GRASP research project include Larry Barowski, Karl Mathias, and Joseph Teate.

About the Authors

James H. Cross II is a professor and chairman of computer science and engineering at Auburn University. He teaches undergraduate and graduate courses in software engineering and directs research in software methods, quality assurance, testing, metrics, and reverse engineering. In particular, he is continuing the development of software engineering courses in design methods and software environments. His research efforts include the GRASP and QUEST/Ada projects, which have received funding from NASA, DISA, and ARPA. He has over 40 refereed publications.

Computer Science and Engineering
107 Dunstan Hall
Auburn University, AL 36849-5347
Voice: 334-844-4330
Fax: 334-844-6329
E-mail: cross@eng.auburn.edu
Internet: http://www.eng.auburn.edu/grasp

Kai H. Chang is an associate professor of computer science and engineering at Auburn University. He is primarily interested in teaching undergraduate and graduate courses in artificial intelligence and expert systems and directing research in expert systems, software quality assurance, testing, metrics, and computer-supported cooperative work environments. His continuing research efforts include the QUEST/Ada project and the Distributed Collaborative Writing Aid project, which has received funding from the National Institute of Standards and Technology. He has over 30 refereed publications.

T. Dean Hendrix is an assistant professor of computer science and engineering at Auburn University. His research interests include software methods, software metrics, reverse engineering, software visualization, and programming languages, including his work with the GRASP project. His teaching interests include courses in software engineering and database systems. He has over 15 refereed publications.

Richard O. Chapman is an assistant professor of computer science and engineering at Auburn University. His research interests include high-level synthesis, hardware-software co-design, and formal methods for hardware and software verification. Chapman's teaching interest are undergraduate and graduate courses in Very Large-Scale Integration computer-aided design, tool design, formal semantics, compilers, and operating systems. His continuing research in high-level synthesis and hardware-software co-design is funded by a CAREER award from the National Science Foundation. He has over 15 refereed publications.

Patricia A. McQuaid is an assistant professor of management information systems at California Polytechnic State University at San Luis Obispo. She has taught a wide range of courses in both the colleges of Business and Engineering. Her research interests include software quality and software testing, particularly in the area of complexity metrics and is the developer of the Profile Metric. She has industry experience in computer auditing and is a certified information systems auditor. She has been invited to speak on the Profile Metric at national and international conferences and has over 15 refereed publications.

Management Information Systems Area
California Polytechnic State University
San Luis Obispo, CA 93407

Voice: 805-756-5381
Fax: 805-756-1473
E-mail: pmcquaid@calpoly.edu

References

1. Cross, J.H., "Improving Comprehensibility of Ada with Control Structure Diagrams," *Proceedings of the Software Technology Conference*, Salt Lake City, Utah, April 11-14, 1994.
2. Cross, J. H., K. H. Chang, and T. D. Hendrix, "GRASP/Ada95: Visualization with Control Structure Diagrams," *Crosstalk, STSC,* Hill Air Force Base, Utah, January 1996, pp. 20-24.
3. Booch, Grady, *Object-Oriented Analysis and Design with Applications,* 2nd ed., Addison-Wesley, Menlo Park, Calif. 1994.
4. McQuaid, P.A., K.A. Chang, and J.H. Cross, "The Profile Metric: A Complexity Metric to Improve Software Quality," *Proceedings of the Fifth European Conference on Software Quality*, Dublin, Ireland, Sept. 16-20, 1996.
5. Baecker, R. M. and A. Marcus, *Human Factors and Typography for More Readable Programs*, ACM Press, 1990.
6. Backer, R., C. DiGiano, and A. Marcus, "Software Visualization for Debugging," *Communications of the ACM*, Vol. 40, No. 4, April 1997, pp. 44-54.
7. Petre, M., "Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming," *Communications of the ACM*, Vol. 38, No. 6, 1995, pp. 33-44.
8. Price, B. A., R. M. Baecker, and I. S. Small, "A Principled Taxonomy of Software Visualization," *Journal of Visual Languages and Computing*, Vol. 4, No. 3, 1993, pp. 211-266.
9. Barnes, J. G. P., *Programming in Ada,* 2nd ed., Addison-Wesley, Menlo Park, Calif., 1984.
10. IEEE Standard VHDL Language Reference Manual 1076-1993, IEEE Computer Society Press, Los Alamitos, Calif., 1993.
11. Basili, Victor and Richard Selby, "Comparing the Effectiveness of Software Testing Strategies," *IEEE Transactions on Software Engineering*, December 1987, Vol. SE-13, No.12, pp.1278-96.
12. Ebenau, Robert, "Predictive Quality Control with Software Inspections," *Crosstalk,* STSC, Hill Air Force Base, Utah, June 1994, pp.9-16.
13. Knight, John C. and B. Littlewood, "Critical Task of Writing Dependable Software," *IEEE Software*, Vol.11, No.1, January 1994, pp.16-20.