Originally published in CrossTalk, Jan. 1996.

GRASP/Ada 95:

Visualization with Control Structure Diagrams

Dr. James H. Cross II, Dr. Kai H. Chang, and T. Dean Hendrix,

Auburn University

Abstract

The Graphical Representations of Algorithms, Structures, and Processes for Ada (GRASP/Ada) project has successfully created and prototyped a new algorithmic level graphical representation for Ada software, the Control Structure Diagram (CSD). The primary impetus to create and refine the CSD is to improve the comprehension efficiency of Ada software, and as a result, improve reliability and reduce costs. The emphasis is on the automatic generation of the CSD from Ada 95 source code to support design, implementation, testing, and maintenance. The CSD has the potential to replace traditional pretty-printed Ada source code. An important additional focus of the GRASP/Ada 95 project is on the generation of a new fine-grained complexity metric called the Complexity Profile Graph (CPG), which will be synchronized with the CSD to provide both visualization and measurement of Ada 95 source code. By synchronizing the CSD and the CPG, the CSD view of control structure, nesting, and source code will be directly linked to the corresponding visualization of statement level complexity in the CPG. In this article, an overview of the GRASP/Ada 95 project at Auburn University is presented with emphasis on the Control Structure Diagram and the current prototype.

Introduction

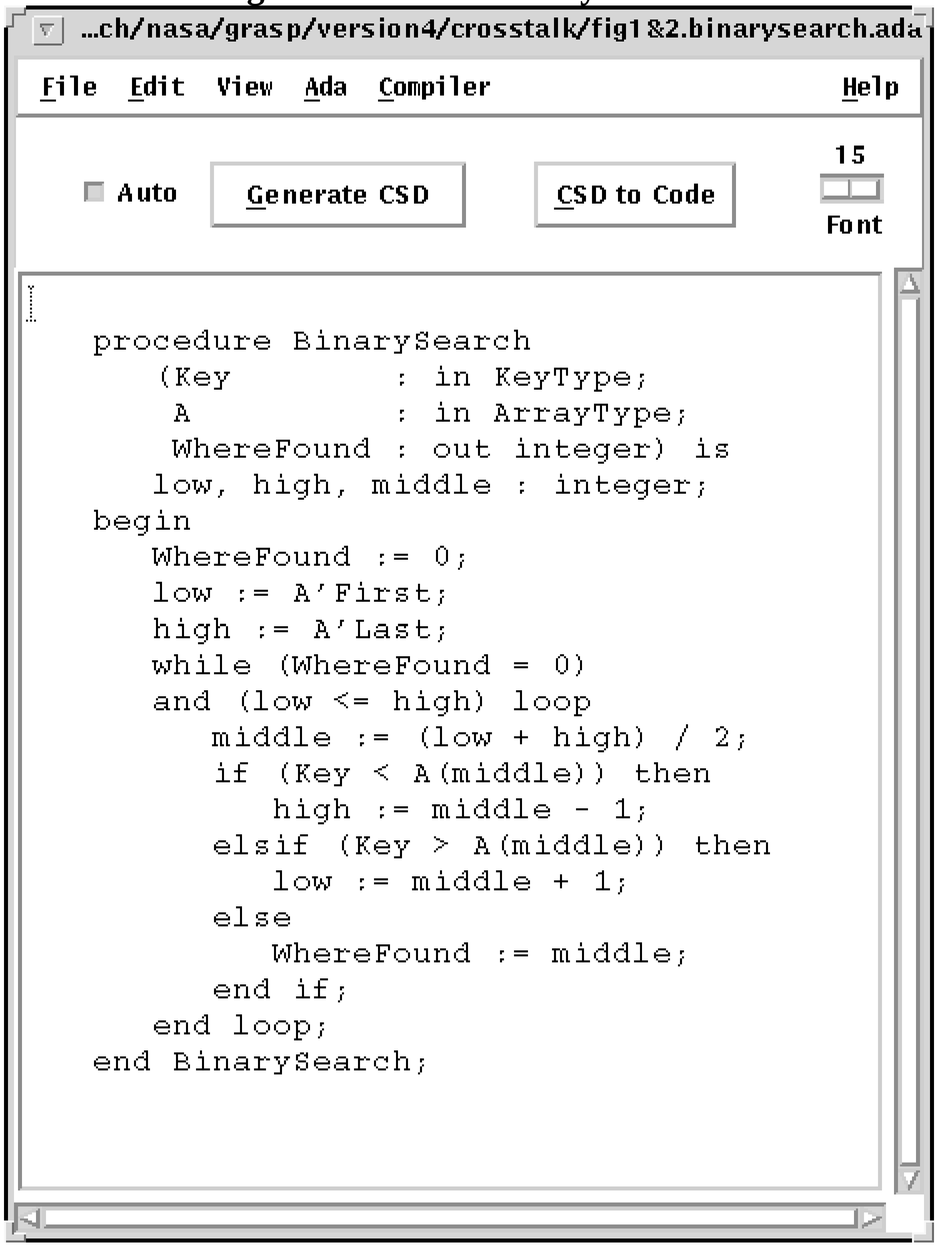
Computer professionals have long promoted the idea that graphical representations of software can be extremely useful as comprehension aids when used to supplement textual descriptions and specifications of software, especially for large, complex systems [1, 2, 3, 4, 5, 6]. The general goal of the GRASP/Ada research project is the investigation, formulation, and generation of *graphical representations of algorithms, structures, and processes for Ada* [7, 8]. The research is currently focused on the generation or reverse engineering of CSDs and CPGs from Ada 95 source code. The CPG is the visualization of a new fine-grained complexity metric [9]. The CSD and the CPG will be synchronized so that 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 primary motivation for the generation of graphical representations is increased support for software lifecycle activities that range from design through 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. The GRASP/Ada 95 software tool has the potential to be a powerful aid in any environment where Ada 95 is expected to be read or written. 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. The following sections describe the control structure diagram and the GRASP/Ada 95 prototype.

The Control Structure Diagram

Although much of the recent research activity in software visualization and computer-aided software engineering tools has focused on architectural-level charts and diagrams, the complex nature of the control constructs and control flow defined by a programming language such as Ada 95 makes source code and detailed design specifications attractive candidates for graphical representation. In particular, source code should benefit from the use of an appropriate graphical notation since it must be read many times during the course of initial development, testing, and maintenance. The CSD is a notation intended specifically for the graphical representation of algorithms in detailed designs as well as actual source code. The primary purpose of the CSD is to reduce the time required to comprehend software by clearly depicting the control constructs and control flow at all relevant levels of abstraction. The CSD is a natural extension to existing architectural graphical representations such as data flow diagrams, structure charts, and object diagrams. The CSD, which was initially created for Pascal [10], has been extended significantly so that the graphical constructs of the CSD map directly to the constructs of Ada [11]. The rich set of control constructs in Ada, e.g., task rendezvous, and the wide acceptance of Ada by the software engineering community as a detailed design language made Ada a natural choice for the basis of a graphical notation. A major objective in the philosophy that guided the development of the CSD was that the graphical constructs should supplement the code or the program design language without disrupting its familiar appearance; that is, the CSD should appear to be a natural extension to the Ada constructs, and similarly, the Ada source code should appear to be a natural extension of the diagram. This has resulted in a concise, compact graphical notation that attempts to combine the best features of diagraming with those of well-indented source code.

Figure 1: Ada for Binary Search.


…ch/nasa/grasp/version4/crosstalk/fig1&2.binarysearch.ada
File Edit View Ada Compiler Help
Auto
Generate CSD
CSD to Code
15
Font

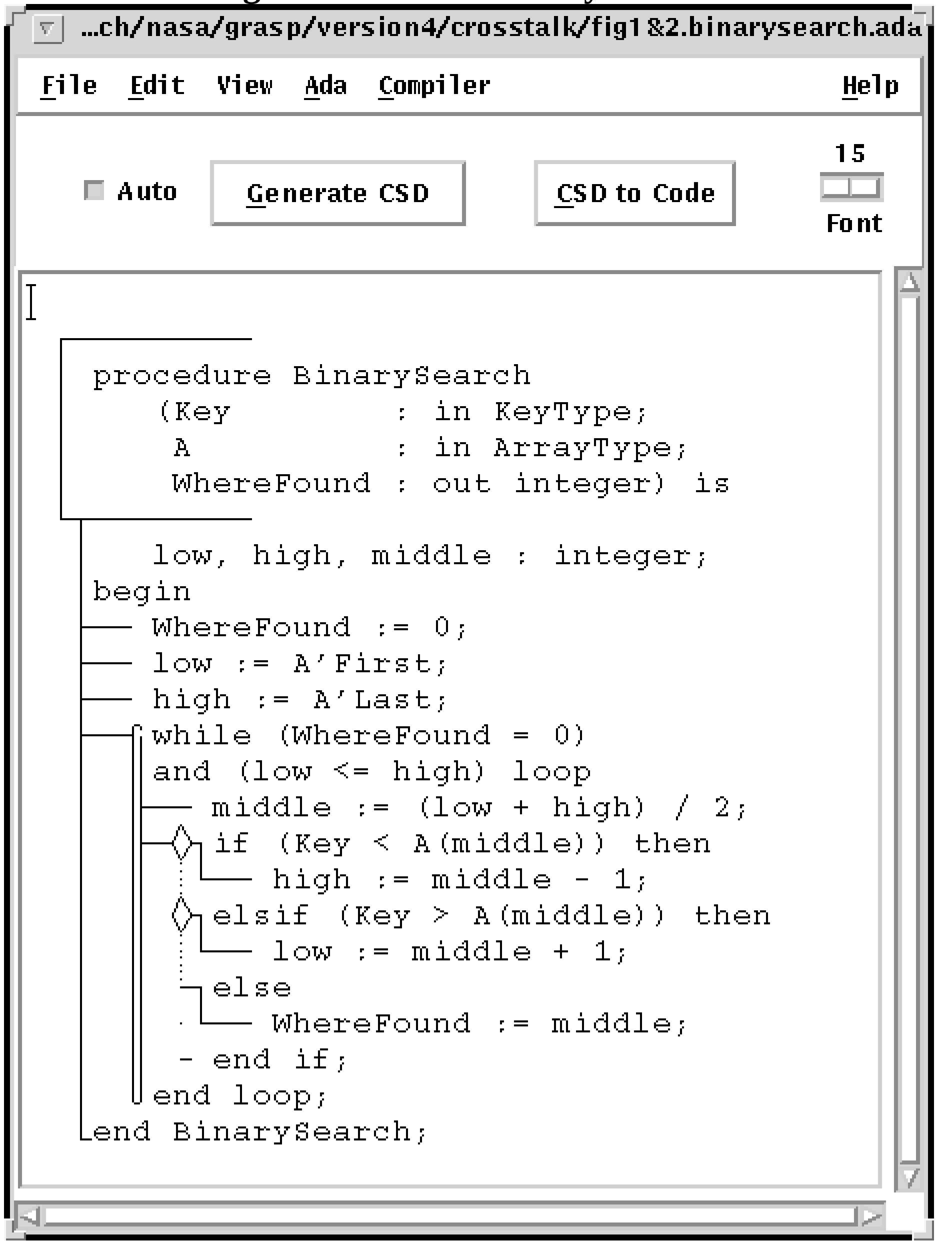
```
procedure BinarySearch
   (Key        : in KeyType;
    A          : in ArrayType;
    WhereFound : out integer) is
   low, high, middle : integer;
begin
   WhereFound := 0;
   low := A'First;
   high := A'Last;
   while (WhereFound = 0)
   and (low <= high) loop
      middle := (low + high) / 2;
      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;
```

Two examples, using the GRASP/Ada 95 CSD generator/editor, are presented in the following figures to illustrate the CSD. The first example shows the basic control constructs of sequence, selection, and iteration that are common to all structured procedural languages such as Ada and C. The second example illustrates a more complex control constructthe task rendezvous in Ada.

Figure 1 contains an Ada procedure called BinarySearch that searches for Key in an array A over its index range-constrained type integer. If Key is found, WhereFound is set to the index of Key in A; otherwise, WhereFound is zero when the procedure terminates.

Figure 2 contains the corresponding CSD. Although this is a simple example, the CSD clearly indicates the levels of control inherent in the nesting of control statements. For example, at level 1 there are four statements executed in sequencethe three assignment statements and the *while* loop. The *while* loop defines a second level of control which contains an assignment statement and an *if* statement, which in turn defines three separate third-level sequences, each of which contains one assignment statement. It is noteworthy that the CSDs for most well-structured production-strength procedures rarely contain more than 10 statements at level 1 or in any of the subsequences defined by control constructs for selection and iteration. This graphical chunking based on functionality and level of control appears to have a substantial positive effect on detailed comprehension of the software. By clicking on the Generate CSD or CSD to Code buttons, users can easily switch between the code and the CSD to make their own assessment of improved readability provided by the CSD.

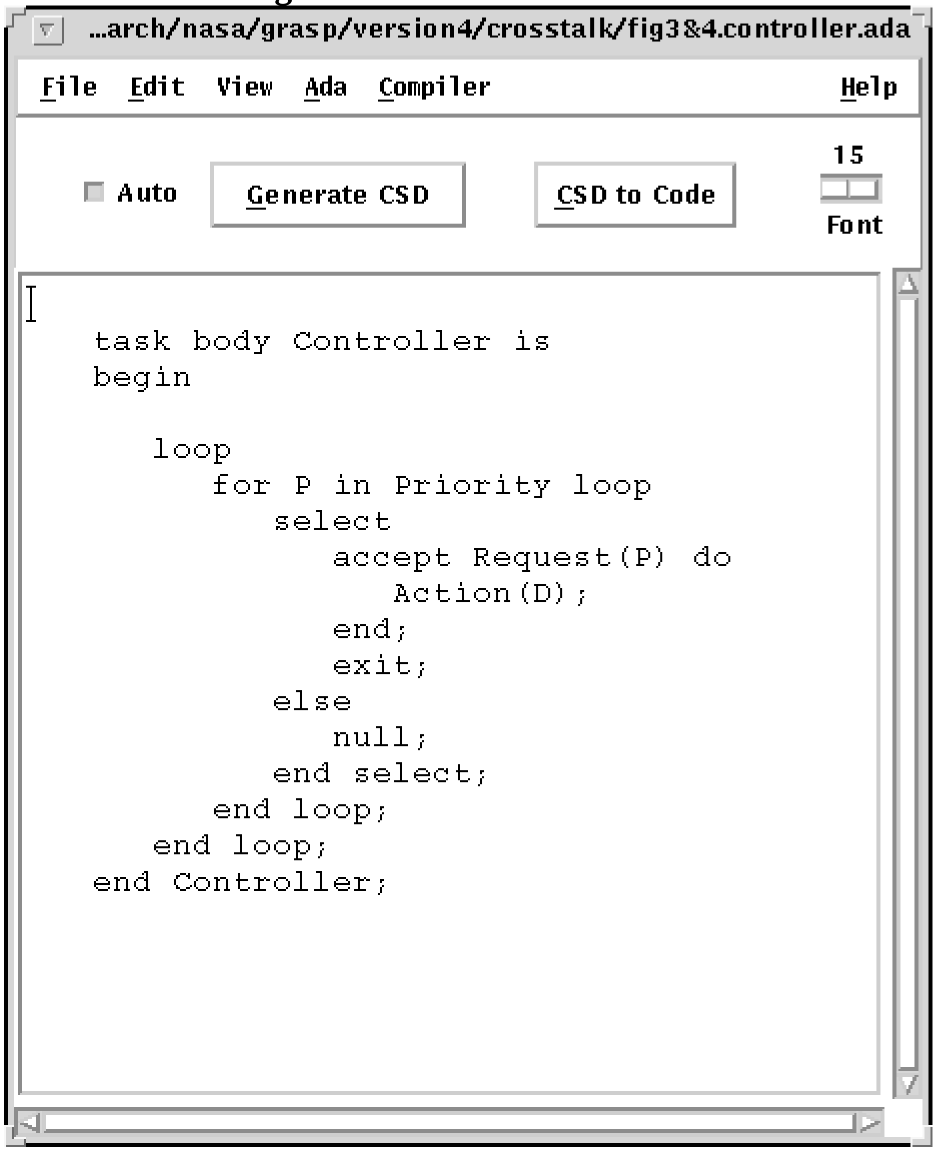
Figure 2: CSD for Binary Search.



```
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: Ada for Controller.

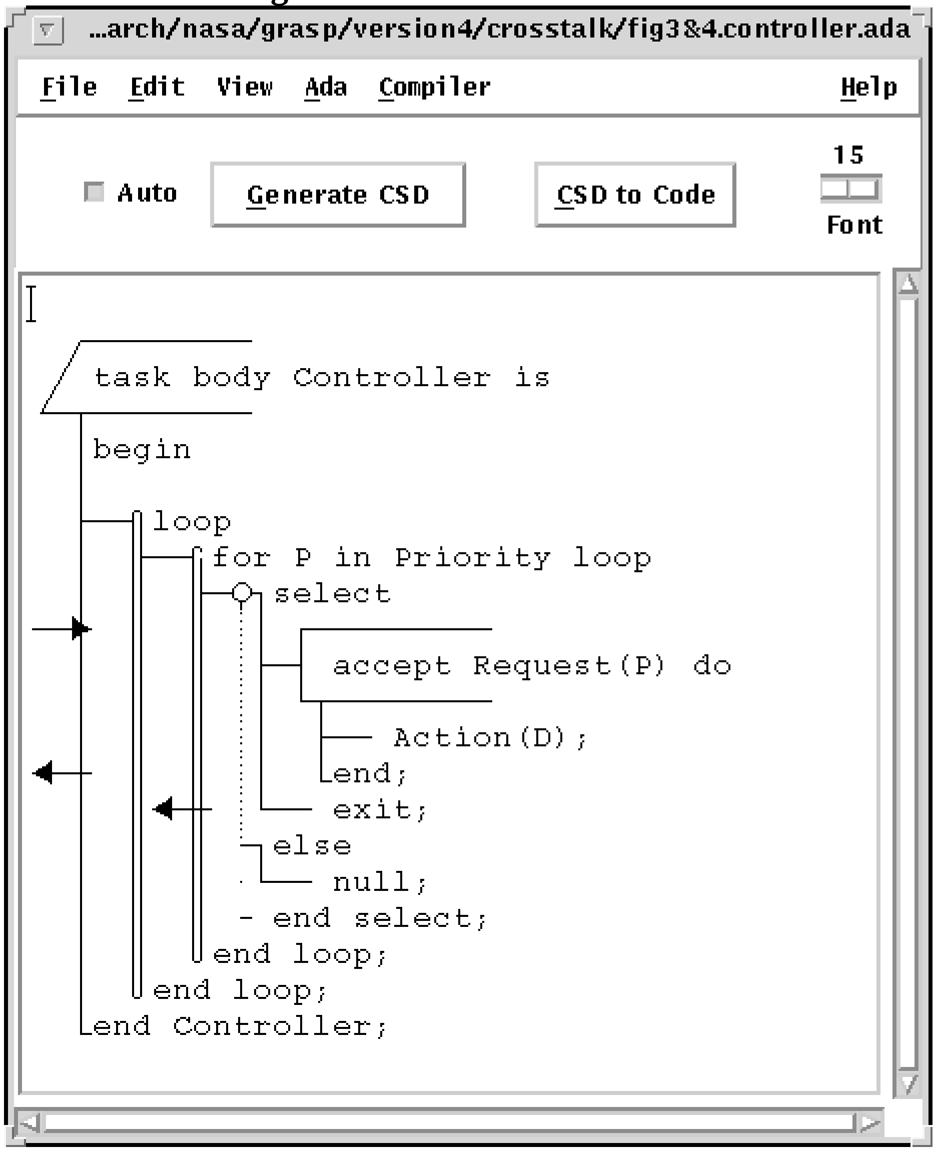


```
task body Controller is
begin

   loop
      for P in Priority loop
         select
            accept Request(P) do
               Action(D);
            end;
            exit;
         else
            null;
         end select;
      end loop;
   end loop;
end Controller;
```

Figure 3 shows an Ada task body controller, which 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 with a task from outside controller, 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. Although the concurrency and multiple exits are useful in modeling the solution, they clearly increase the effort required of the reader to comprehend the code.

Figure 4: CSD for Controller.



```
task body Controller is
begin
   loop
      for P in Priority loop
         select
            accept Request(P) do
               Action(D);
            end;
            exit;
         else
            null;
         end select;
      end loop;
   end loop;
end Controller;
```

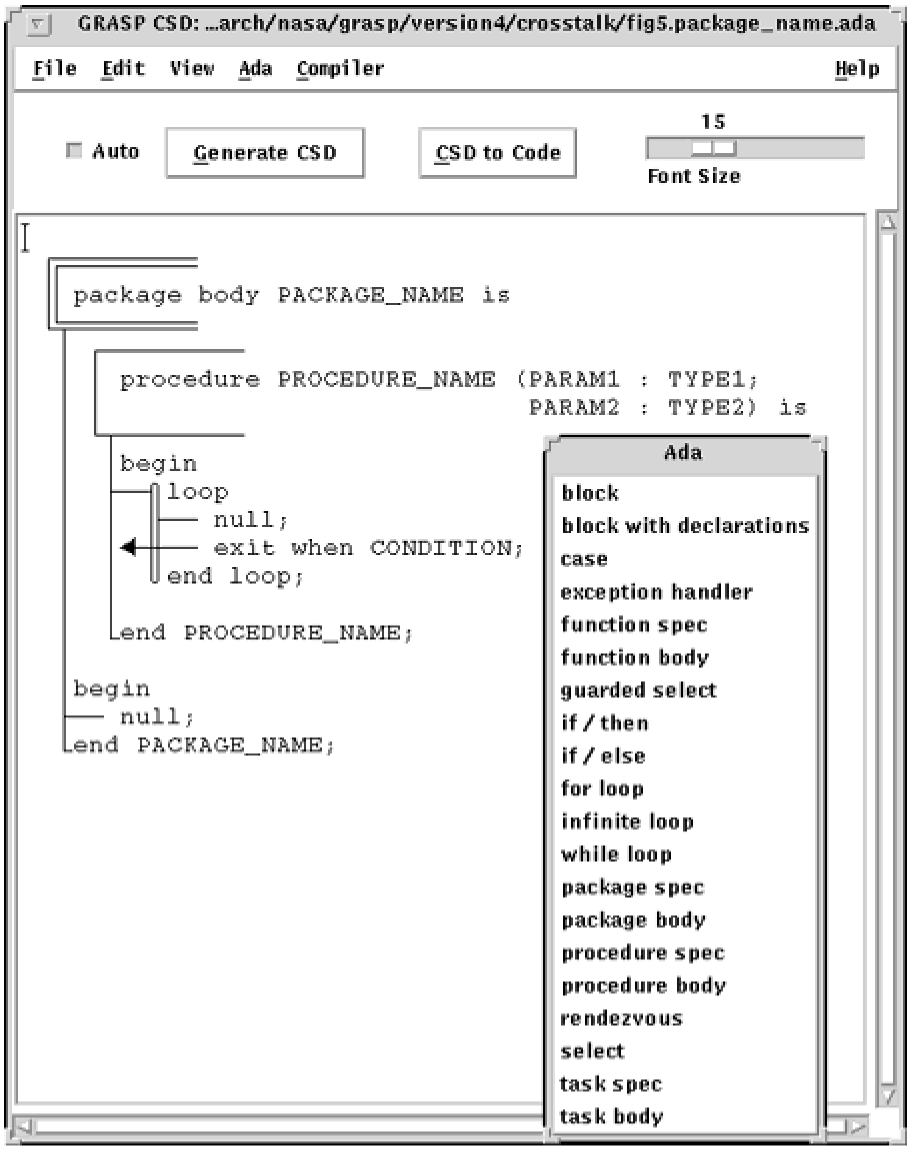
The CSD in Figure 4 uses intuitive graphical constructs to depict the point of rendezvous, the two nested loops, the select statement enclosing the accept statement for the task, the unconditional exit from the inner loop, and the overall control flow of the task. When reading the code without the diagram, as shown in Figure 3, the control constructs and control paths are much less obvious, although the same structural and control information is indicated by indentation and the semantics of the text. 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. In fact, our experience to date indicates that after reading source code with a CSD as in Figure 4, users have a definite preference for it over source code only as in Figure 3.

Ongoing research includes creation of additional graphical constructs as deemed appropriate and the creation of custom font symbols for each program unit in Ada 95. The program unit symbols are a new and innovative feature that is expected to increase the effectiveness of the CSD by providing a visual link from the CSD to *object diagrams* at the architectural level as well as the more traditional structure charts and data flow diagrams.

The GRASP/Ada Prototype

The current prototype, Version 4.2, provides a CSD window, which is a full-function text editor with the capability to generate, display, edit, and print CSDs. The File and Edit options are similar to traditional text editors. View will allow the user to select between the current CSD drawing mode and the future program unit symbols. The Ada option opens a tear-off menu of selectable Ada templates. When a template name is clicked, it is inserted at the point of the cursor. The CSD in Figure 5 is the result of clicking on templates for *package body*, *procedure bod*y, and *infinite loop*. The user may add custom templates to the list.

Figure 5: Using Ada Templates.



```
package body PACKAGE_NAME is

   procedure PROCEDURE_NAME (PARAM1 : TYPE1;
                             PARAM2 : TYPE2) is

   begin
      loop
         null;
         exit when CONDITION;
      end loop;

   end PROCEDURE_NAME;

begin
   null;
end PACKAGE_NAME;
```

Version 4.3 (March 1996) will provide for coupling with an Ada compiler. The CSD window will allow the user to invoke an Ada compiler directly for the current program unit, and when an error is reported by the compiler, the offending line of code will be highlighted in the diagram. Version 4.3 beta is currently being tested as a front end for GNAT [12] in three computer science and engineering courses at Auburn University.

Since the CSD generation and display cycle is extremely fast (less than one-half second for 3,000 lines of Ada), no residual intermediate files are written under normal operation. When a file that contains an Ada program unit is loaded, the CSD is automatically generated if Auto (next to Generate CSD on the tool bar) is turned on. Otherwise, the user may generate the CSD on demand by clicking the Generate CSD button, which is usually done routinely during the course of editing to redraw the diagram. If a parse error is encountered during CSD generation, the cursor is moved to the highlighted line that contains the error to aid the user in making corrections. When the user saves a file, the CSD is filtered so that only the Ada source code is retained. The net result is that the CSD window can be used in place of a traditional program editor to generate, display, edit, and print CSDs with no additional overhead, i.e., the CSD is essentially free.

```
                            _______________
                           |               |
                           |  GUI (Motif)  |
                           |_______________|
                          /        |         \
                        /          |           \
                      /            |             \
                    /              |               \
                  /                |                 \
            ____/_____         ____|____           ____\____
           |          |       |         |         |         |
           |  CSDgen  |       |  CPGen  |         |  ODgen  |
           |__________|       |_________|         |_________|
                |                  |                   |
           _____|__________________|___________________|_____
          |           GRASPlib  UNIX File System             |
          |__________________________________________________|
             |              |              |              |
             |              |              |              |
             |              |              |              |
        source code        CSD            CPG            OD
```

Figure 6: GRASP/Ada 95 Block Diagram

The major system components of the GRASP/Ada 95 prototype, which is being written in a combination of C and Ada 95, are shown in the block diagram in Figure 6. The user interface was built using Motif and the X Window System and includes a main control window to provide general coordination among the other componentsCSDgen, CPGgen, and ODgen. Version 5 will include the generation of the complexity profile graph (CPGgen) and a CPG window, which will be synchronized with the current CSD window. Version 6 will include an object diagram generation component, ODgen, and respective object diagram window. Automatic diagram layout is under investigation, and several design alternatives have been identified, including whether to integrate GRASP/Ada directly with commercial components. For example, GRASP/Ada could be integrated with an Ada development environment that provides an Ada semantic interface specification that would support the identification and extraction of objects. The GRASP/Ada library component, GRASPlib, supports coordination of all generated items with their associated source code. The current file organization uses standard UNIX directory conventions as well as default naming conventions to facilitate navigation among the diagrams and the production of sets of diagrams.

Conclusion

The CSD has the potential to significantly improve the quality of detailed design and implementation by supplementing or replacing traditional pretty-printed Ada source code. As illustrated by Figures 1 through 5, the CSD provides a compact graphical representation. When compared to traditional source code listings, the CSD requires little, if any, additional space for storage and hard copy. The additional time required to generate a CSD using GRASP/Ada, display it in a window, or print it using a laser printer is negligible.

The graphical constructs of the CSD map directly to the constructs of Ada 95. The rich set of control constructs in Ada 95, e.g., task rendezvous, and the wide acceptance of Ada by the software engineering community as a detailed design language made Ada a natural choice for the basis of a graphical notation. A major objective in the philosophy that guided the development of the CSD was that the graphical constructs should supplement the code without disrupting its familiar appearance; that is, the CSD should appear to be a natural extension to the Ada constructs, and similarly, the Ada 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 diagraming with those of well-indented source code.

The current GRASP/Ada prototype, although only one of a set of required visualization tools, has clearly indicated the utility of the CSD. Enhancements to the CSD and the addition of the CPG and OD will only increase its effectiveness as a tool to improve the comprehensibility and measurement of software.

Acknowledgments

Research on the GRASP/Ada project has been supported in part by grants from Marshall Space Flight Center, and the Advanced Research Projects Agency. Current graduate students working on the GRASP/Ada prototype include Larry Barowski and Joseph Teate.

James H. Cross II, Kai H. Chang, and T. Dean Hendrix
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

References

1. Martin, J., and C. McClure, *Diagramming Techniques for Analysts and Programmers*, Englewood Cliffs, N.J., Prentice-Hall, 1985.

2. Shu, Nan C., *Visual Programming*, New York, N.Y., Van Norstrand Reinhold Company, Inc., 1988.

3. Aoyama, M., et. al., "Design Specification in Japan: Tree-Structured Charts," *IEEE Software*, March 1989, pp. 31-37.

4. Scanlan, D.A., "Structured Flowcharts Outperform Pseudocode: An Experimental Comparison," *IEEE Software*, September 1989, pp. 28-36.

5. Tripp, L.L., "A Survey of Graphical Notations for Program DesignAn Update," *ACM Software Engineering Notes*, Vol. 13, No. 4, 1989, pp. 39-44.

6. Cross, J.H., E.J. Chikofsky, and C.H. May, "Reverse Engineering," *Advances in Computers*, Vol. 35, 1992, pp. 199-254.

7. Cross, J.H., "Improving Comprehensibility of Ada with Control Structure Diagrams," *Proceedings of Software Technology Conference*, April 11-14, 1994, Salt Lake City, Utah (distributed on CD-ROM), 25 pages.

8. Cross, J.H., and T.D. Hendrix, "Using Generalized Markup and SGML for Reverse Engineering Graphical Representations of Software," *Proceedings of Working Conference on Reverse Engineering*, July 16-19, 1995, Toronto, pp. 2-6.

9. McQuaid P.A., K.H. Chang, and J.H. Cross, "Complexity Metric to Aid Software Testing and Maintenance," *Proceedings of Decision Sciences Institute*, Nov. 20-22, 1995, Boston, Mass., Vol. 2, pp. 862-864.

10. Cross, J.H., and S.V. Sheppard, "The Control Structure Diagram: An Automated Graphical Representation For Software," *Proceedings of the 21st Hawaii International Conference on Systems Sciences* (Kailui-Kona, Hawaii, Jan. 5-8, 1988), IEEE Computer Society Press, Washington, D.C., 1988, Vol. 2, pp. 446-454.

11. Cross, J.H., S.V. Sheppard, and W.H. Carlisle, "Control Structure Diagrams for Ada," *Journal of Pascal, Ada, and Modula 2*, Vol. 9, No. 5, September/October 1990.

12. "Introduction to GNAT," *Release Documents for GNAT Version 2.07*, New York University, July 16, 1995.

About the Authors

Dr. James H. Cross II is an associate professor of computer science and engineering at Auburn University. Cross is primarily interested in teaching undergraduate and graduate courses in software engineering and directing research in software methodology, quality assurance, testing, metrics, and reverse engineering. In particular, Cross is continuing the development of software engineering courses in design methodology and software environments. His continuing research efforts include the GRASP/Ada and QUEST/Ada projects, which are funded by National Aeronautics and Space Administration and the Advanced Research Projects Agency. The GRASP/Ada project has focused on reverse engineering and, in particular, the automatic generation of graphical representations of software. The QUEST/Ada project has focused on a rulebased approach to the automatic generation of test cases. Cross has over 30 refereed technical publications.

Dr. Kai H. Chang is an associate professor of computer science and engineering at Auburn University. Chang 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. Chang has over 30 refereed technical publications.

T. Dean Hendrix is a doctorate candidate in computer science and engineering at Auburn University and an instructor of computer science at Jacksonville State University. Hendrix's dissertation topic is markup languages for graphical representations. He is primarily interested in teaching courses in software engineering and doing research in the areas of programming languages, software methodology, metrics, and reverse engineering.