
Empirical Software Engineering, 3, 131–158 (1998)

The Control Structure Diagram: An Overview and Initial Evaluation

JAMES H. CROSS II AND T. DEAN HENDRIX
Computer Science and Engineering, Auburn University, USA

SAEED MAGHSOODLOO
Department of Industrial and Systems Engineering, Auburn University, USA

Abstract. A new graphical representation, the Control Structure Diagram (CSD), has been created to visualize software at both the source code and program design language (PDL) level. The primary impetus for creation of the CSD was to improve the comprehension efficiency of software and, as a result, improve reliability and reduce costs. The CSD has the potential to replace traditional prettyprinted source code. As part of the GRASP (Graphical Representations of Algorithms, Structures, and Processes) research project at Auburn University, the GRASP software engineering tool has been successfully developed. GRASP automatically generates CSDs from source code written in Ada, C, C++, Java, and VHDL. The emphasis to this point has been on the automatic generation of the CSD to support development, maintenance, reverse engineering and reengineering through the use of GRASP. GRASP has been applied successfully to numerous programs ranging in size from several hundred to several thousand lines of source code and is efficient and sufficiently flexible for use in a production setting. To demonstrate the potential benefits of the CSD and its automatic generation using GRASP, a series of empirical studies has been planned and initiated. First, as reported in this article, the perceived usefulness of the CSD was evaluated using a preference instrument based on eleven performance characteristics in which a comparison was made with other well-known graphical representations for algorithms. Statistical analysis indicated numerous significant differences with a clear preference for the CSD in seven of the eleven performance characteristics. Further empirical studies, currently being implemented, will examine the effect of the CSD and GRASP on objective measures such as comprehension efficiency and effectiveness.

Keywords: Ada, graphical representation, software visualization, empirical evaluation, reverse engineering

1. Introduction

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 (Aoyama, 1989; Baecker et al., 1997; Baecker and Marcus, 1990; Petre, 1995; Scanlan, 1989; Shu, 1988). While the general goal of the GRASP research project has been the investigation, formulation and generation of graphical representations of algorithms, structures, and processes (GRASP) at several levels of abstraction, the current focus has been on the automatic generation or reverse engineering of control structure diagrams (CSDs) from source code using the GRASP tool.

Reverse engineering normally includes analyzing source code to extract higher levels of abstraction for both data and processes. The primary motivation for reverse engineering is increased support for software reusability, verification, software maintenance and reengineering, all of which should be greatly facilitated by automatically generating a set of formalized diagrams to supplement source code and other forms of existing documentation. An overall goal of current software engineering research is to provide the foundation for

software engineering environments in which reverse engineering and forward engineering (development) are tightly coupled. In this environment, the user is assured that graphical representations accurately reflect the software in one of two ways: (1) the user may specify the software in a graphically-oriented language and then automatically generate the corresponding source code or (2) the user may specify the software in source code or PDL and then automatically generate the graphical representations either dynamically as the code is entered or as a form of post-processing. The focus in the GRASP project has been on the latter alternative.

The following sections describe the control structure diagram, the GRASP system, and an evaluation of the CSD. Cross et al. (1990) and Cross (1994) describe the overall rationale for the development of the CSD, and Chikofsky and Cross (1990) and Cross et al. (1992) provide a taxonomy and extensive literature review of reverse engineering. Cross et al. (1996) describes the GRASP tool and its use.

2. The Control Structure Diagram

The complex nature of the control constructs and control flow defined by many programming languages and their associated PDLs, makes source code and detailed design specifications attractive candidates for visualization. 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 control structure diagram (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 companion to existing architectural graphical representations such as data flow diagrams, structure charts, and object diagrams.

The CSD, which was initially created for Pascal/PDL (Cross and Sheppard, 1988), has been extended significantly so that all the control constructs of Ada, C, C++, Java, and VHDL are directly represented as graphical constructs in the CSD. A major objective in the philosophy that guided the development of the CSD was that the graphical constructs should supplement the code and/or PDL without disrupting their familiar appearance. That is, the CSD should appear to be a natural extension to 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 PDL or source code.

2.1. Background

Graphical representations have been recognized as having an important impact in communicating from the perspective of both the writer and the reader. For software, this includes communicating requirements between users and designers and communicating design specifications between designers and implementers. However, there are additional areas where the potential of graphical notations has not been fully exploited. These include communicat-

ing the semantics of the actual implementation represented by the source code to personnel for the purposes of testing and maintenance, each of which consumes major resources in the software life cycle. In particular, Selby (1985) found that code reading was the most cost effective method of detecting errors during the verification process when compared to functional testing and structural testing. Standish (1985) reported that program understanding represents a large portion of maintenance cost. Hence, improved comprehension efficiency resulting from the integration of graphical notations and source code could have a significant impact on the overall cost of software production.

Since the flowchart was introduced and studied in the 1940s, numerous notations for representing algorithms have been proposed and utilized. Several authors have published notable books and papers that address the details of many of these (Martin and McClure, 1985; Price et al., 1993; Raeder, 1985). Tripp (1989), for example, describes 18 distinct notations that have been introduced since 1977 and Aoyama (1989) describes popular diagrams used in Japan. In general, these diagrams have been strongly influenced by structured programming and thus contain control constructs for sequence, selection, and iteration. In addition, several contain explicit exit structures to allow single entry / multiple exit control flow through a block of code, as well as parallel or concurrency constructs. However, none of the diagrams cited explicitly contains all of the control constructs found in a control-rich language such as Ada.

Although there are numerous graphical representations of source code available and automated tools are increasing in availability and function, the use of graphical representations of software has been neglected by business and industry in the U.S. in favor of non-graphical PDL. Several factors contribute to this neglect. First, simply generating a graphical representation of software is not sufficient. There is an element of secondary notation (Petre, 1995) that is difficult to automate and yet is crucial to providing cognitive leverage to the programmer. Unless an automated tool pays close attention to issues such as secondary notation, the resulting graphical representation may not be particularly useful. Second, even when graphical representations are carefully prepared, their usefulness is not clear. There are several studies (Green et al., 1991; Green and Petre, 1992; Moher et al., 1993; Petre, 1995) which cast doubt on the value of graphical representations in some cases. Finally, professional programmers have not, in general, shown a preference for using much of the available program visualization technology. This could be due to a variety of factors and should not necessarily be seen as negative toward the use of graphical representations in general. Clearly, however, more research is needed in this area to bring the potential benefits of the technology to professional programmers.

Recent results of the GRASP research project give encouragement that these important issues are being addressed. Empirical studies reported in Aoyama (1989), Scanlan (1989), Petre (1995) have concluded that graphical notations may indeed improve the comprehensibility and overall productivity of software. The study reported by Scanlan (1989) involved a well-controlled experiment in which deeply nested if-then-else constructs, represented in structured flowcharts and pseudo-code, were read by intermediate-level students. Scores for the flowchart were significantly higher than those of PDL. The statistical studies reported by Aoyama (1989) involved several tree-structured diagrams (e.g. PAD, YACC II, and SPD) widely used in Japan which, in combination with their environments, have led

```
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 tasking example.

to significant gains in productivity. The results of these and other studies suggest potential benefits to be gained from the effective use of graphical representations.

2.2. The Control Structure Diagram Illustrated

The power of the CSD in program comprehension can be seen by comparing Figure 1 with Figure 2. Each figure contains the same Ada task body adapted from Barnes (1984). This task loops through a priority list attempting to selectively accept 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.

The CSD in Figure 2, automatically generated by GRASP, uses intuitive graphical constructs to depict the point of rendezvous, the two nested loops, the select statement guarding the accept statement for the task, the unconditional exit from the inner loop, and the overall control flow of the task. When reading the plain source code in Figure 1, the control constructs and control paths are much less visible although the same structural and control information is available. With additional levels of nesting and increased physical separation

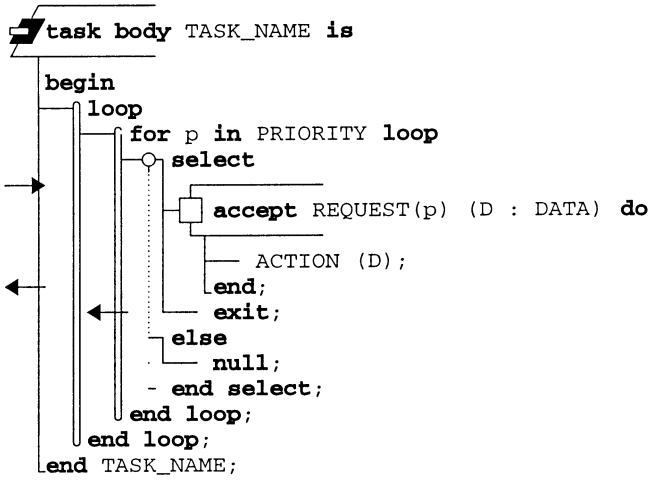


Figure 2. Ada tasking example rendered as CSD.

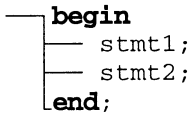


Figure 3. CSD sequential control.

of sequential components, the visibility of control constructs and control paths becomes increasingly obscure in the absence of an explicit visual aid such as the CSD.

Now that the CSD has been briefly introduced, the primary CSD constructs are presented in more detail. For illustration purposes, Ada was chosen as the underlying source language to be used in all the examples. Sequential control is represented by the simplest CSD construct, the vertical line. Individual statements on a particular level of sequential control are marked with small horizontal lines attached to the vertical line for that level (see Figure 3).

Selection in Ada is accomplished through one of four basic control constructs: the if statement, the if-else statement, the if-elsif statement, and the case statement. The CSD notation for each of these is shown in Figure 4. The condition for selection is marked with a small diamond, just as in a flowchart. The statements to be executed if the condition is true are marked by a solid line leading from the right of the diamond. The control path for a false condition is marked with a dotted line leading from the bottom of the diamond to

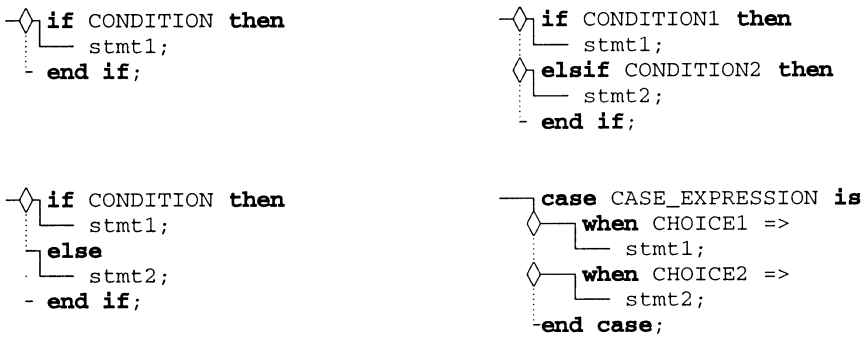


Figure 4. CSD selection constructs.

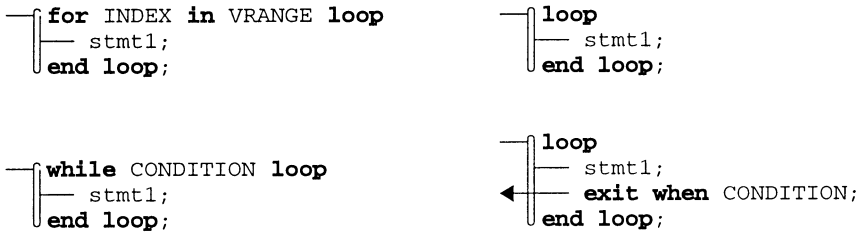


Figure 5. CSD iteration control.

the next executable statement, either another decision diamond, an else clause, an others clause, or the end of the decision statement.

Iteration in Ada is provided by three looping constructs: the for loop, the while loop, and the basic loop as shown in Figure 5. The CSD looping symbol is open at the top for the while loop and for loop, denoting the point at which iteration may end in these pretest loops. The CSD for the basic loop is unbroken since this loop is infinite by nature. An exit statement may be used in the basic loop to terminate the iteration. The exit statement is denoted in the CSD by an arrow which points to the control level where the program should continue its execution after the loop is exited. When the exit statement indicates a specific loop (by name), the exit arrow is drawn to the appropriate level.

The CSD uses five different types of open-ended boxes to identify the major Ada program units: the single box, the double box, the slanted box, and the single and double boxes with dashed lines. Each particular box represents a specific group of Ada program units, as illustrated in Figure 6. The single box is used to identify both specifications and bodies for

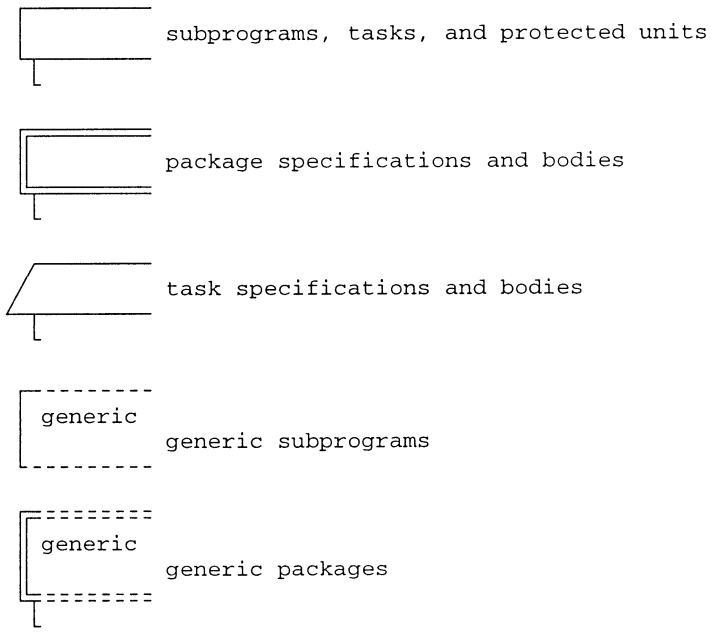


Figure 6. CSD Box Notation.

Ada subprograms, protected types, exception handlers, and task entries. The double box is used to identify both package specifications and package bodies in Ada. The slanted box is used to identify the body and specification of the Ada task. Dashed single boxes identify generic subprograms and dashed double boxes identify generic packages.

CSD unit symbols, illustrated in Figure 7, provide the user with the option of specializing the program unit notation. These unit symbols are patterned after Booch's module notation (Booch and Bryan, 1994) but include additional original symbols such as protected specification and body, and exception handler. As programs increase in size and complexity, the CSD unit symbols become more useful in comprehending the Ada source code since they can provide a direct visual relation to the architectural diagrams of the system.

3. GRASP

Unless graphical representations such as the CSD can be automatically generated from source code in an efficient and robust manner, it is unlikely that they will see use in practice. GRASP is a prototype software engineering tool built to automatically generate CSDs from source code written in Ada, C, C++, Java, and VHDL. The major system components of

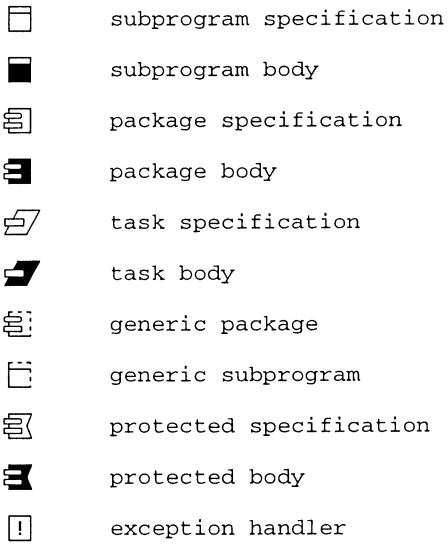


Figure 7. CSD Unit Symbols.

the current release of the GRASP prototype are shown in the block diagram in Figure 8.

The CSDgen component controls the generation of control structure diagrams and the CPGgen component controls the generation of complexity profile graphs (McQuaid et al., 1995). Although not yet completed, the ODgen component controls the generation of object diagrams. The GRASP library component, GRASPlib, supports coordination of all generated items with their associated source code.

The user controls and organizes a GRASP session through a single control panel. The control panel allows the user to set tool properties such as text font and color, as well as open one or more CSD windows. Each CSD window, as depicted in Figure 9, is a full-function text editor with the capability to generate, display, edit, and print CSDs for source code files. In addition to visualization and editing capabilities, the CSD window also provides compilation, linking, and execution facilities for all supported languages.

The current CSDgen prototype builds the CSD according to the language independent model for software visualization reported in (Cross and Hendrix, 1996). The implementation is quite efficient, even for industrial strength use, generating CSDs at approximately 5,000 lines of code per second on a Sun Sparc20.

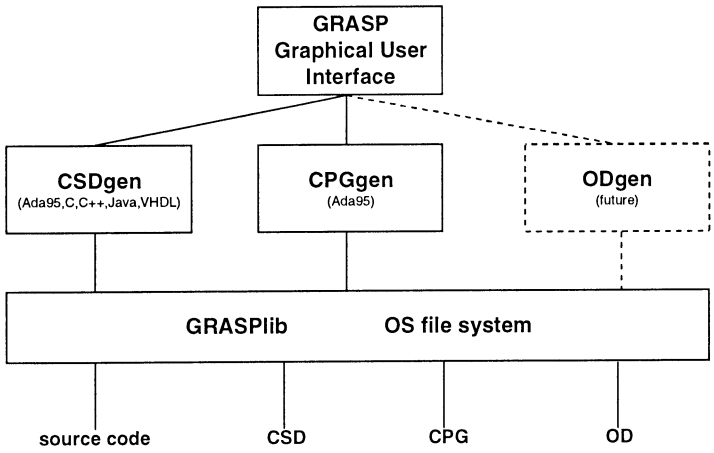


Figure 8. GRASP system model.

4. Evaluation of the Control Structure Diagram

An important aspect of any research project is evaluation of results. Formal statistically-based controlled experiments dealing with the comprehensibility of graphical representations of software are difficult to design and conduct although several such experiments have been reported in the literature (Green et al., 1991; Green and Petre, 1992; Petre, 1995). Similar difficulties are encountered when attempting to design controlled experiments to evaluate software engineering tools with respect to improvements in productivity that result from their use (Marshall et al., 1998). The primary difficulty arises from the learning curve that users/subjects must overcome. For example, a year or more may be required to become proficient enough with a software tool to actually realize gains in productivity. Thus, it may be difficult to compare two software tools in a controlled experiment without introducing bias based on familiarity or in many cases the lack of familiarity. As a result, most evaluation of software tools is based on preference surveys in which the user/subject is asked to make mental assessments or comparisons of various aspects of the tool(s) under study.

Although subjective user preference of a tool is not necessarily a good predictor of objective performance gains when using that tool (Kissel, 1995), careful evaluation of this measure can yield significant useful results. It is clear from industry experience reports that regardless of the objective qualities and advantages of a tool, it will not be used in practice without the willingness and motivation of users. Hence, from a pragmatic viewpoint, user perception and preference play a significant role in the actual use of new tools and techniques.

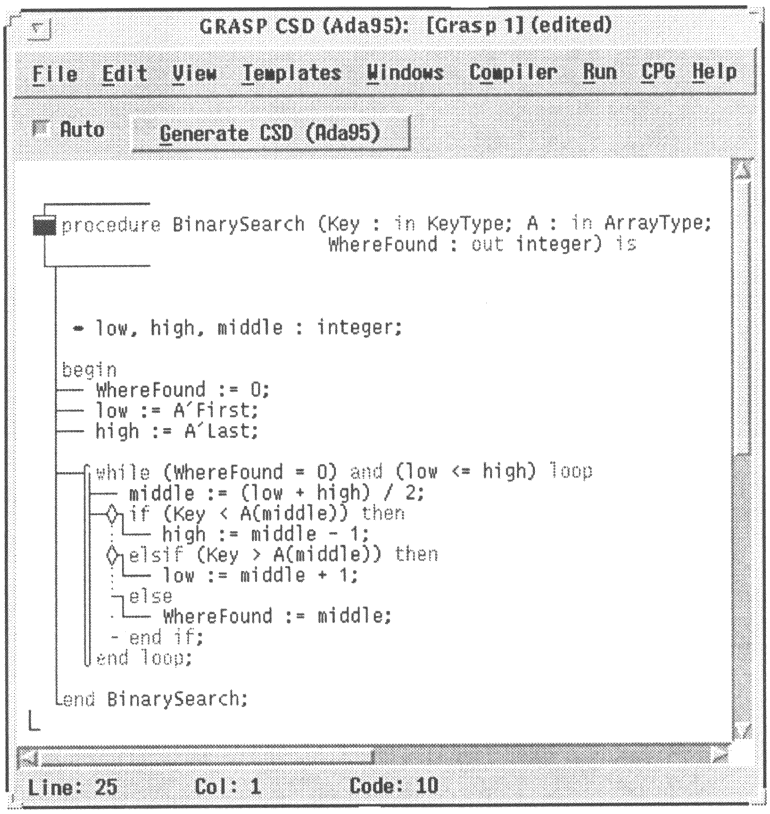


Figure 9. GRASP CSD Window.

The evaluation of the CSD and GRASP is being done in two phases. The first, reported in this article, is an evaluation of the preference of the CSD relative to other popular graphical representations. User perception and preference are significant elements in a systematic study and evaluation of the CSD, and are important companions to objective performance measures. The second phase will be an empirical evaluation of such an objective performance measure—the effect of the CSD on comprehension of software. This second phase experiment will be designed at least at the level of the work reported by Curtis et al. (1989) and similar experiments. The preference evaluation was performed first due to its timeliness relative to the proposed second phase comprehension experiment. The results of the preference evaluation were expected to provide motivation for continuing (or not) to the second phase.

This section describes the design of the first phase experiment, the subjects that partic-

ipated in the evaluation of the CSD, the preference survey instrument that was developed and administered, and the results of data analysis.

4.1. The Design of the Experiment

The primary objective of the evaluation was to determine preference, if any, for the CSD over other similar graphical representations for algorithms. The ANSI flowchart (FC), the Nassi-Shneiderman chart (NS), the Warnier-Orr diagram (WO), and the action diagram (AD), were selected for comparison due to their similarity in purpose to the CSD and their high degree of familiarity and use in the field. Figure 10 illustrates each of the graphical representations used in the experiment.

The experiment was set up as a block design in which each subject was considered a block and each diagram type (FC, NS, WO, AD, CSD) represented a treatment. The subjects were to compare the treatments with respect to eleven performance characteristics (PCHs).

On the evaluation instrument, the first three items solicited background information with respect to familiarity with the five diagram types. The next eleven items indicated PCHs by which the subjects were asked to compare the diagrams with respect to (1) how well each represented sequence, (2) selection, and (3) iteration, (4) overall readability, (5) improvement in readability as an extension to pseudo-code, (6) ease of coding from, (7) ease of manual use, (8) overall preference if drawn manually, (9) overall economy, (10) overall preference with equivalent automated support, and finally (11) overall preference all assumptions aside. These eleven items are described in more detail below in the discussion of results. The instrument concluded with an open ended question soliciting suggestions on how to improve any of the diagrams compared. Indeed, we have made several improvements to the CSD since this experiment was performed. Where details of the CSD are presented in this article, the most recent form is discussed.

4.2. The Subjects

The evaluation instrument was administered to 33 junior/senior computer science and engineering students at Auburn University in the course CSE 422—Introduction to Software Engineering, during a single quarter. These students all had experience with FORTRAN, Pascal, and C in previous courses. None had formal training in Ada. Since participation in the evaluation was optional, five bonus points to be added to the final exam score were offered as an incentive. All students present took part in the evaluation.

4.3. Preparation of the Subjects

Prior to the experiment, as part of the normal agenda of the course, the students were presented with the mechanics of each graphical notation used in the experiment and then required to use all five notations to represent portions of the source code or PDL of their course software project. Thus, at the time of the experiment, the students were responding

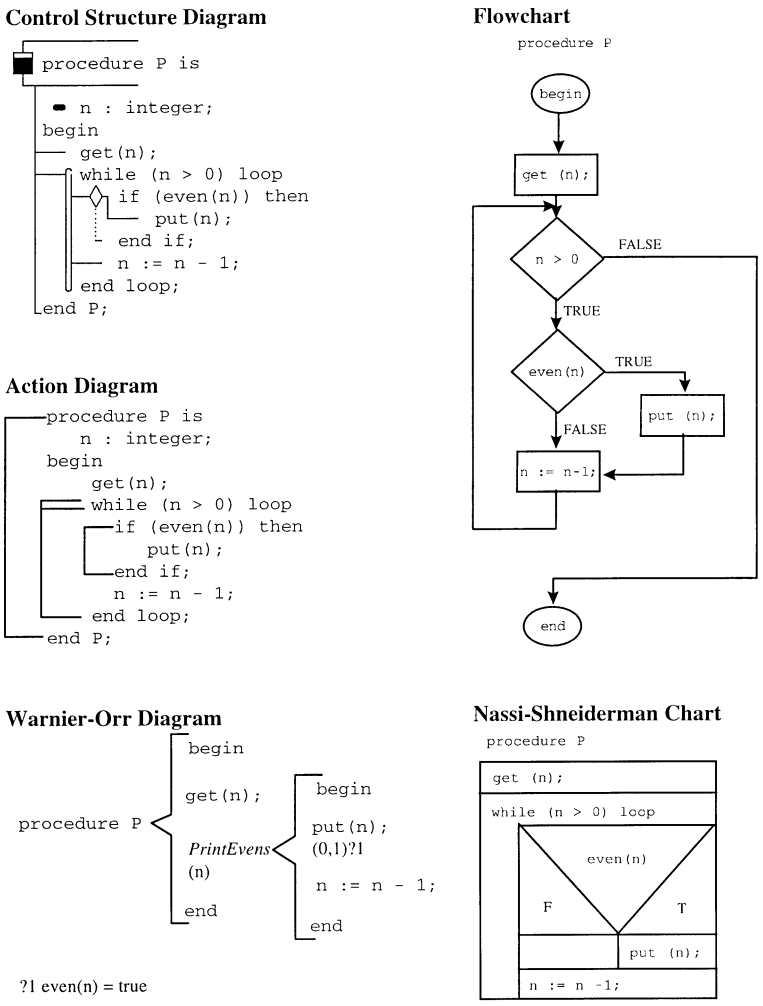


Figure 10. Illustration of the five graphical representations.

to the evaluation instrument based on their memories of the five notations and their use with their own code, not simply contrived exercises and examples. This preparation was designed to give the students more realistic experiences with the use of each graphical representation and to strengthen the results of the evaluation.

Great care was taken to eliminate any opportunity for the instructor's association with GRASP and the CSD to introduce bias into the evaluation. The CSD was presented as just another graphical representation, without any reference to its origins or the author.

Absolutely no connection between the CSD and the students' instructor was implied at anytime during the course. Such deliberate care was taken not only to strengthen the results of the evaluation, but also for personal reasons. At the time of the experiment, the author remained unconvinced that the CSD would be perceived as being useful. Thus, it was desirable that no association between the CSD and its creator be indicated to the students.

4.4. The Evaluation Results

An item analysis was performed on the data collected for the eleven PCHs in the first part of the evaluation instrument (i.e., all items except the three background items at the beginning and the last item which asked for suggested improvements to the diagrams). Following the item analysis, an analysis of variance was performed to determine if differences in the preferences were statistically significant. The results of item analysis and the tests for significance are presented below, followed by a general summary of the responses from the second part of the evaluation instrument.

4.4.1. Item Analysis of Comparison of Graphical Representations

The subjects were given the following instructions regarding the eleven performance characteristics.

```
Based on the experience you have gained by using these
diagramming tools to represent algorithms, you are asked to
assign a rating to each of the diagrams with respect to a
specific comparison among the diagrams. You may assign the
same rating to more than one diagram for a given comparison.
Select your ratings from the following scale and enter them
as indicated below.

   5 - best / most / first choice
   4 -
   3 - moderate
   2 -
   1 - worst / least / last choice

For each of the eleven items below, the subjects used the
rating scale above to rank the following treatments:
(FC, NS, WO, AD, CSD)

1.  Compare the diagrams with respect to how well each shows
    sequence.
2.  Compare the diagrams with respect to how well each shows
    selection.
3.  Compare the diagrams with respect to how well each shows
```

iteration.
4. Compare these diagrams with respect to overall readability (consider reading someone else's code).
5. Each of these tools can be used with informal pseudocode as opposed to actual statements in a programming language and, as such, can be thought of as a graphical extension to pseudocode (with possibly some spatial rearrangement). Rate the diagrams on the extent to which they increase readability over non-graphical pseudocode.
6. Suppose, as a programmer, you are given a design specification in which the program logic has been documented using one of the graphical representations below. Compare the diagrams with respect to which would best facilitate your task of coding from the design specification.
7. Compare the diagrams with respect to ease of manual use; consider the initial drawing and subsequent modifications.
8. Assuming you have to manually draw the diagrams (in the sense that they are not automatically generated), indicate your overall preference for each diagram where: 5 - first choice, . . ., 1 - last choice.
9. Compare the diagrams with respect to their overall economy (i.e., increases in comprehension versus effort to draw them manually).
10. Assuming you have equivalent automated support to draw each of the diagrams in the sense that the diagrams are automatically generated either by selecting constructs from a menu or by parsing the code, indicate your overall preference for each diagram where: 5 - first choice, . . ., 1 - last choice.
11. All assumptions aside, indicate your overall preference for each diagram.

The analysis results for the above eleven items are summarized in Tables 1 and 2. Table 1 shows the number of responses from the 33 students for each PCH and diagram type (FC, NS, WO, AD, CSD). A number less than 33 indicates the performance characteristic for that particular diagram type was left blank. Students were advised orally to leave an item blank if they were unfamiliar with the notation or a particular construct. Note that 20 students responded to all performance characteristics for all five diagrams, and 32 students responded to all performance characteristics for three of the diagrams (FC, NS, CSD). Table 2 contains the averages for the responses computed on the basis of only those items completed.

The results in Table 2 clearly indicate the overall trend in preference for the CSD. For an additional perspective, the averages in Table 2 were converted into percentages (out of a maximum possible of rank 5) and are shown in Table 3 [1]. The combined PCH averages of these percentages are depicted in Figure 11.

Table I. Item response frequencies (The number of students who responded to a particular treatment.)

PCH	FC	NS	WO	AD	CSD
1. SEQ	33	33	25	22	33
2. SEL	33	33	24	21	33
3. ITR	33	33	24	21	33
4. GEN READ	33	32	24	20	33
5. EXT P-CODE	33	33	24	21	33
6. CODE- FROM	33	32	25	22	32
7. MANUAL	33	32	23	24	33
8. PREF/MANL	33	33	26	25	33
9. ECONOMY	33	33	25	24	33
10. PREF/AUTO	33	33	25	23	33
11. PREF/GEN	33	33	26	25	33

Table II. Item analysis for graphical representations (all responses).

PCH	FC	NS	WO	AD	CSD
1. SEQ	3.21	3.64	2.64	2.32	3.94
2. SEL	3.52	4.06	2.46	2.05	3.64
3. ITR	3.45	3.48	2.58	2.14	3.91
4. GEN READ	3.03	3.38	2.67	2.10	4.24
5. EXT P-CODE	2.85	3.76	2.38	2.48	3.94
6. CODE- FROM	2.82	3.53	2.60	2.14	4.31
7. MANUAL	3.09	3.16	2.61	2.38	3.91
8. PREF/MANL	3.00	3.30	2.42	2.16	4.15
9. ECONOMY	2.70	3.27	2.52	2.00	4.52
10. PREF/AUTO	3.03	3.52	2.36	2.09	4.55
11. PREF/GEN	3.00	3.33	2.54	1.96	4.55
PCH Average	3.06	3.49	2.53	2.17	4.15

Table 4 shows the difference between the control structure diagram (CSD) percentage scores and each of the other percentage scores. Positive values indicate a preference for the CSD, and negative values indicate a preference for the indicated diagram type. Note that the NS SEL (selection) construct was the only item for which the CSD construct was not preferred on average. Item 5 is of particular interest in that it attempts to determine perceived improvements in readability over non-graphical pseudo-code.

Tables 1 through 4 provide useful insight and suggest potential significant differences among the preferences. However, an analysis of variance is required to determine the presence or absence of actual statistical significance.

4.4.2. Statistical Analysis for Significance of Preference of Graphical Representations

Since the data were taken on a scale of 1 to 5 and thus were not continuous, the parametric Analysis of Variance (ANOVA) could not be applied. Note that ANOVA generally assumes

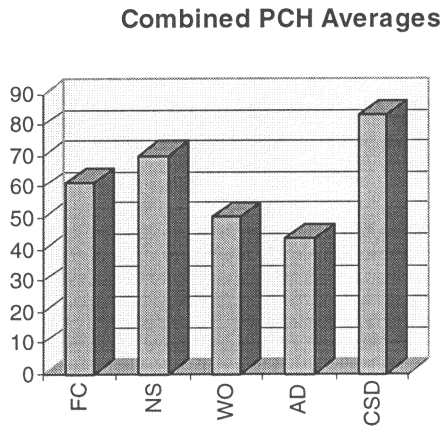


Figure 11. Combined PCH averages.

Table III. Percentage scores for graphical representations (all responses).

PCH	FC	NS	WO	AD	CSD
1. SEQ	64.20	72.80	52.80	46.40	78.80
2. SEL	70.40	81.20	49.20	41.00	72.80
3. ITR	69.00	69.60	51.60	42.80	78.20
4. GEN READ	60.60	67.60	53.40	42.00	84.80
5. EXT P-CODE	57.00	75.20	47.60	49.60	78.80
6. CODE- FROM	56.40	70.60	52.00	42.80	86.20
7. MANUAL	61.80	63.20	52.20	47.60	78.20
8. PREF/MANL	60.00	66.00	48.40	43.20	83.00
9. ECONOMY	54.00	65.40	50.40	40.00	90.40
10. PREF/AUTO	60.60	70.40	47.20	41.80	91.00
11. PREF/GEN	60.00	66.60	50.80	39.20	91.00
PCH Average	61.27	69.87	50.51	43.31	83.02

that the data originates from a normal population—an assumption that would not be tenable in this case. Therefore, a nonparametric (or distribution-free) test was used to determine if there were statistically significant differences among the five treatments FC, NS, WO, AD, and CSD. The determination of significant differences would be made with respect to all eleven PCHs.

Each student responded to at least 3 of 5 treatments (i.e., each student ranked at least 3 of the treatments FC, NS, WO, AD, and CSD). Therefore, each student is considered as a block and the relevant model is the randomized complete block design. The appropriate nonparametric

Table IV. Percentage score differences: CSD—others.

PCH	FC	NS	WO	AD	CSD
1. SEQ	14.60	6.00	26.00	32.40	
2. SEL	2.40	−8.40	23.60	31.80	
3. ITR	9.20	8.60	26.60	35.40	
4. GEN READ	24.20	17.20	31.40	42.80	
5. EXT P-CODE	21.80	3.60	31.20	29.20	
6. CODE- FROM	29.80	15.60	34.20	43.40	
7. MANUAL	16.40	15.00	26.00	30.60	
8. PREF/MANL	23.00	17.00	34.60	39.80	
9. ECONOMY	36.40	25.00	40.00	50.40	
10. PREF/AUTO	30.40	20.60	43.80	49.20	
11. PREF/GEN	31.00	24.40	40.20	51.80	
PCH Average Diff.	21.75	13.15	32.51	39.71	

ANOVA procedure is the Friedman's Test (Conover, 1980). This test analyzes ranked data for a complete randomized block design. Since only 20 students responded to all 5 treatments, initially the test of significance was performed for differences among all five treatments. The null hypothesis is as follows:

H_0 : There are no significant differences among the 5 treatments;

versus the alternative:

H_1 : There are significant differences (between at least two of the treatments).

For the sake of illustration, the data for the PCH SEQ is provided in Table 5.

Table 5 shows that, for example, student number 23 ranked FC, NS, WO, AD, CSD as 4, 2, 3, 3, and 5, respectively. The Friedman's Test requires that the responses for all treatments be ranked within each block from 1 to 5, and therefore, the sum of the ranks within each block is:

$$\sum_{j=1}^{5} R_{ij} = \sum_{j=1}^{5} j = 15$$

The tied ranks receive average ranks. The last five columns (FC_R, NS_R, WO_R, AD_R, and CSD_R) of Table 9 give the ranks for each of the 20 students for the performance characteristic SEQ after tied ranks have been averaged. Now the ranks assigned by student number 23 become 4.0, 1.0, 2.5, 2.5, and 5.0 respectively. Since the student ranked NS the lowest, the treatment NS_R received a rank of 1.0. The student ranked WO and AD equally and next lowest, therefore ranks 2 and 3 were averaged and WO_R and AD_R each received 2.5. Similarly, FC_R received a rank of 4.0 and CSD a rank of 5.0. Thus, we have

$$\sum_{j=1}^{5} R_{13,j} = 4.0 + 1.0 + 2.5 + 2.5 + 5 = 15$$

as expected for student number 23 in row 13. The 2-way (treatments and blocks) ANOVA is conducted on ranks as illustrated below.

Table V. Data for performance characteristic SEQ (PCH #1, for 20 students).

Stu#	Student Responses					Assigned Ranks				
	FC	NS	WO	AD	CSD	FC_R	NS_R	WO_R	AD_R	CSD_R
2	1	2	5	4	3	1.0	2.0	5.0	4.0	3.0
3	5	1	2	4	3	5.0	1.0	2.0	4.0	3.0
7	5	4	4	2	3	5.0	3.5	3.5	1.0	2.0
8	5	3	2	1	4	5.0	3.0	2.0	1.0	4.0
11	5	3	3	3	5	4.5	2.0	2.0	2.0	4.5
14	3	3	1	2	4	3.5	3.5	1.0	2.0	5.0
15	5	4	3	3	3	5.0	4.0	2.0	2.0	2.0
16	4	2	3	1	5	4.0	2.0	3.0	1.0	5.0
17	3	4	2	2	3	3.5	5.0	1.5	1.5	3.5
19	1	4	2	2	5	1.0	4.0	2.5	2.5	5.0
20	1	5	4	2	3	1.0	5.0	4.0	2.0	3.0
21	3	3	3	3	3	3.0	3.0	3.0	3.0	3.0
23	4	2	3	3	5	4.0	1.0	2.5	2.5	5.0
25	1	5	3	2	4	1.0	5.0	3.0	2.0	4.0
27	3	4	1	2	5	3.0	4.0	1.0	2.0	5.0
29	4	5	1	2	3	4.0	5.0	1.0	2.0	3.0
30	1	2	3	4	5	1.0	2.0	3.0	4.0	5.0
31	5	4	2	1	3	5.0	4.0	2.0	1.0	3.0
32	2	5	3	1	4	2.0	5.0	3.0	1.0	4.0
33	2	5	1	3	4	2.0	5.0	1.0	3.0	4.0
					Totals ($R_{\cdot j}$)	63.5	69.0	48.0	43.5	76.0

Let $SS(Total)$, $SS(Treatments)$, $SS(Blocks)$, and $SS(Residuals)$ represent total sum of squares, treatment sum of squares, block sum of squares, and residual sum of squares, respectively. Then

$$\begin{aligned} SS(Total) &= \sum_{i=1}^{20}\sum_{j=1}^{5} R_{ij}^2 - \frac{300^2}{100} \\ &= 1^2 + 2^2 + 5^2 + \cdots + 4^2 - 900 \\ &= 182.50 \end{aligned}$$

where the correction factor is

$$CF = \frac{(\sum_{i=1}^{20}\sum_{j=1}^{5} R_{ij})^2}{100} = \frac{(\sum_{i=1}^{20}(15))^2}{100} = \frac{300^2}{100}$$

$$\begin{aligned} SS(Treatments) &= \sum_{j=1}^{5}\left(\frac{R_{\cdot j}^2}{20}\right) - 900 \\ &= \frac{63.5^2 + 69.0^2 + 48.0^2 + 43.5^2 + 76.0^2}{20} - 900 \\ &= 38.275 \end{aligned}$$

Table VI. ANOVA.

Source	df Degrees of Freedom	SS Sum of Squares	MS Mean Squares $= SS/df$	F_0 $(SS_{tr}/4)/$ $(SS_{tr}/76)$
Total	99 $= \#Tr(\#Stu) - 1$	182.5		
Treatments	4 $= \#Tr - 1$	38.275	9.56875	5.0423
Blocks	19 $= \#Stu - 1$	0	0	
Residuals	76 $= Totl - Tr - Blk$ $= 99 - 4 - 19$	144.225	1.8977	

$$
\begin{aligned}
SS(Blocks) &= \sum_{i=1}^{20} \frac{(R_{i.})^2}{5} - \frac{300^2}{100} \\
&= \frac{15^2 + 15^2 + \cdots + 15^2}{5} - 900 \\
&= \frac{20(225)}{5} - 900 = 0
\end{aligned}
$$

Note that, since each block subtotal is 15, then

$$SS(Blocks) = SS(Students)$$

is identically zero. Hence, we have the following:

$$
\begin{aligned}
SS(Residual) &= SS(Total) - SS(Treatments) - SS(Blocks) \\
&= 144.225
\end{aligned}
$$

The results of the ANOVA are shown in Table 6. Having computed the F statistic, we must now determine if it is sufficiently large to reject the null hypothesis. Since the 1 percentage point of the F distribution with 4 and 76 degrees of freedom (df) is $F_{.01}(4, 76) = 3.577$ which is less than $F_0 = 5.0423$, we reject the null hypothesis that there were no significant differences among the FC, NS, WO, AD, and CSD with respect to the performance characteristic "Sequence."

Now that the null hypothesis has been rejected, we have either made a correct decision or we have committed a Type I error (note that a Type I error is committed when an experimenter rejects a true hypothesis). The probability of committing a Type I error, or the level of significance of the test is determined from $\hat{\alpha} = P(F_{4,76} \geq F_0)$. The probability

(or critical) level of the test in this case is $\hat{\alpha} = 0.00117$ which indicates there is very little probability of a Type I error. Note that the smaller the critical level is, the more strongly H_0 can be rejected and the more significant are the differences among the treatments.

The Friedman's Test was conducted on the complete data set of 20 students for all five treatments and the results are summarized in Table 7. The Friedman's statistic, F_0, showed that, except in the case of performance characteristic MANUAL, the differences among the 5 treatments were highly significant ($\hat{\alpha} \ll 0.01$). In the case of MANUAL, the differences among the five treatments were significant at the level ($\hat{\alpha} = 0.0133$).

Table 2 clearly shows the average ratings that the treatments WO and AD received were lower than the other three, FC, NS, and CSD. Furthermore, 32 students rated FC, NS, and CSD as opposed to only 20 who rated all five treatments. As a result, the Friedman's Test was also applied to determine if the three treatments FC, NS, CSD differed significantly. Again each student's rating of FC, NS, and CSD were ranked as 1, 2, 3 (average ranks were assigned to equal ratings as before) and the Friedman's Test was applied for each of the eleven PCHs. The results, summarized in Table 8, shows that the differences among the three treatments FC, NS, CSD were not significant (at the .05 level) for the four performance characteristics SEQ, SEL, ITR, and MANUAL, but the three treatments differed very significantly with respect to each of the other seven performance characteristics. Furthermore, the average ranks for CSD far exceeded those for FC and NS in the case of the seven significantly different performance characteristics.

5. Analysis of Experimental Results

This experiment is significant and important not because the CSD was generally preferred to the other four graphical representations, but rather because of the answers it suggests to important questions in the design of graphical representations and software visualizations in general. What makes one graphical representation preferable to another? In what manner or for what purposes do users prefer one graphical representation to another? The answers to these questions are central to software visualization research. This experiment suggests answers to these questions that are similar to those described by (Petre, 1995).

(Petre, 1995) describes the notions of primary and secondary notations as being crucial to the usefulness of a graphical representation. The primary notation of a graphical representation is the explicit set of symbols of that notation together with the rules for combining the individual symbols. Secondary notation includes those properties not explicitly in the primary notation but nonetheless useful in composing visualizations with the graphical representation. For example, boxes, diamonds, and arrows are part of the primary notation of flowcharts while proximity and size are elements of that graphical representation's secondary notation.

The differentiation and preference of one graphical representation over another based on issues related to secondary notation is clearly suggested by this experiment. Recall that there were three graphical representations generally preferred over the others: the flowchart, Nassi-Shneiderman chart, and the CSD. When asked directly about these three graphical representations' ability to visualize basic control structures (sequence, selection, and iteration) there was no significant preference among the subjects for one representation

Table VII. ANOVA summary for treatments FC, NS, WO, AD, CSD.

Total Ranks Assigned by 20 students for 5 Treatments

	PCH	FC_R	NS_R	WO_R	AD_R	CSD_R
1.	SEQ	63.5	69.0	48.0	43.5	76.0
2.	SEL	70.5	85.5	39.0	38.0	67.0
3.	ITR	74.0	68.0	50.5	40.5	67.0
4.	GEN READ	63.5	67.0	47.0	39.5	83.0
5.	EXT P-COD	57.0	79.5	41.5	46.5	75.5
6.	CODE-FROM	57.0	70.5	45.5	43.0	84.0
7.	MANUAL	65.0	68.0	48.5	46.0	72.5
8.	PREF/MANL	63.5	69.5	42.5	45.0	79.5
9.	ECONOMY	55.0	71.5	49.0	38.5	86.0
10.	PREF/AUTO	55.0	70.0	42.0	40.5	92.5
11.	PREF/GEN	57.0	70.0	46.0	38.0	89.0

ANOVA Summary for Treatments FC, NS, WO, AD, and CSD $\hat{\alpha} = P(F_{4,76} \geq F_0)$

	PCH	SS_TOTL	SS_TR	MS_TR	SS_RESID	MS_RESID	F_0	$\hat{\alpha}$
1.	SEQ	182.50	38.2750	9.5688	144.2250	1.8977	5.0423	.00117
2.	SEL	188.00	86.7250	21.6813	101.2750	1.3326	16.2703	.08115
3.	ITR	182.50	38.9750	9.7438	143.5250	1.8885	5.1596	.00099
4.	GEN READ	191.50	58.9750	14.7438	132.5250	1.7437	8.4552	.000011
5.	EXT P-COD	185.00	57.7000	14.4250	127.3000	1.6750	8.6119	.00000871
6.	CODE-FROM	194.50	59.7250	14.9313	134.7750	1.7734	8.4198	.0000112
7.	MANUAL	189.50	28.6750	7.1688	160.8250	2.1161	3.3877	.013231
8.	PREF/MANL	196.50	50.7000	12.6750	145.8000	1.9184	6.6070	.000129
9.	ECONOMY	187.00	70.8250	17.7063	116.1750	1.5286	11.5832	.000000215
10.	PREF/AUTO	194.00	94.2750	23.5688	99.7250	1.3122	17.9616	.09202
11.	PREF/GEN	197.00	81.5000	20.3750	115.5000	1.5197	13.4069	.07258

over another. That is, the subjects perceived that all three were equally able to denote the presence of the basic control structures in source code. This result agrees with intuition since this part of the evaluation was based on the graphical representations' primary notation. Since all three graphical representations do offer primary notation to depict sequence, selection, and iteration, it is intuitively clear that the flowchart, Nassi-Shneiderman chart, and the CSD are equally able to indicate the presence of the basic control structures.

Preference for the CSD over the other graphical representations became evident only when the subjects were asked to respond to questions relating to using the graphical representations in certain contexts or for certain tasks. As (Petre, 1995) attests, how well a graphical representation can be used in a certain context or for a certain task depends heavily on appropriate use of secondary notation. Items on the evaluation instrument which revealed a clear preference for the CSD were those which addressed issues such as readability, ease of manual use and modification, and the cost/benefit ratio of using the graphical representation. These are all issues that relate directly to secondary notation and the subtle implicit characteristics of a graphical representation.

The clear preference for the CSD on these items suggests that there are secondary characteristics of the CSD which make it especially readable, modifiable, and economic to

Table VIII. ANOVA Summary for Treatments FC, NS, CSD.

Total Ranks Assigned by 32 Students for 3 Treatments

	PCH	FC_R	NS_R	CSD_R
1.	SEQ	57.0	66.0	69.0
2.	SEL	60.0	73.5	58.5
3.	ITR	64.0	58.0	70.0
4.	GEN READ	54.0	57.0	81.0
5.	EXT P-COD	51.0	68.5	72.5
6.	CODE-FROM	46.5	63.0	82.5
7.	MANUAL	60.0	58.0	74.0
8.	PREF/MANL	53.5	61.0	77.5
9.	ECONOMY	46.0	59.0	87.0
10.	PREF/AUTO	48.5	57.0	86.5
11.	PREF/GEN	50.0	58.0	84.0

ANOVA Summary for Treatments FC, NS, and CSD $\hat{\alpha} = P(F_{2,62} \geq F_0)$

	PCH	SS_TOTL	SS_TR	MS_TR	SS_RESID	MS_RESID	F_0	$\hat{\alpha}$
1.	SEQ	59.00	2.4375	1.2188	56.5625	0.9123	1.3359	0.27
2.	SEL	59.00	4.2656	2.1328	54.7344	0.8828	2.4159	.098
3.	ITR	59.50	2.2500	1.1250	57.2500	0.9234	1.2183	.303
4.	GEN READ	61.00	13.6875	6.8438	47.3125	0.7631	8.9683	0.00004
5.	EXT P-COD	54.00	8.1719	4.0859	45.8281	0.7392	5.5278	0.0062
6.	CODE-FROM	61.50	20.2969	10.1484	41.2031	0.6646	15.2708	.000004
7.	MANUAL	58.00	4.7500	2.3750	53.2500	0.8589	2.7653	.071
8.	PREF/MANL	62.00	9.4219	4.7109	52.5781	0.8480	5.5551	.006
9.	ECONOMY	60.50	27.4375	13.7188	33.0625	0.5333	25.7259	.000000007
10.	PREF/AUTO	61.50	24.8594	12.4297	36.6406	0.5910	21.0324	.0000001
11.	PREF/GEN	64.00	19.7500	9.8750	44.2500	0.7137	13.8362	0.000011

use. Three specific design properties of the CSD are proposed as providing its advantage in secondary notation: The CSD is *intuitive*, *non-disruptive*, and *efficient*. The CSD was designed to use intuitive symbols which suggest their meaning by their appearance. For example, loops in source code are represented by a symbol which physically loops upon itself. The CSD is also made more intuitive by a more direct mapping from its primary notation to source code constructs. For example, there exists a unique symbol for iteration in the CSD while the flowchart depicts iteration as a composite of other symbols. The CSD is non-disruptive. That is, the CSD does not displace the source code as the flowchart and Nassi-Shneiderman chart do. This allows a reader of the CSD to leverage the familiar appearance of source code with the graphical representation. The non-disruptiveness of the CSD also resolves complex layout issues that can exist with other graphical representations. The layout of the CSD is exactly that of the source code. The CSD is also efficient. Since there are no complex layout issues, the CSD can be rendered very quickly from source code, either manually or automatically. Modifications to source code are also easily accommodated in the CSD. Not only is the CSD quite time efficient, it is also space efficient. When the CSD unit symbols (Figure 7) are used, the CSD requires no more space on the printed page or the display screen than well-indented source code, and only marginally

more when using the CSD box notation (Figure 6) or combined units symbols and box notation (Figure 2).

The analysis of the experimental results above indicated that the preference shown to the CSD was due to subtle yet extremely important issues of secondary notation. These results will be used as the basis of the planned second phase evaluation of the CSD, which will include a highly structured objective evaluation of the effect of the CSD on the human comprehension of software. It is expected that future objective evaluations of the CSD will support the results from this experiment.

6. Future Directions

6.1. Control Structure Diagram

The CSD constructs are expected to continue to evolve and increase in usefulness. Current research efforts include designing *abstraction scalability* into the CSD. That is, the CSD should aid the software engineer in comprehension through various levels of abstraction in a natural and seamless manner. For example, the CSD unit symbols provide a visual reference to the architectural level depicted by a module diagram. When the object diagram generation component is implemented, GRASP will provide seamless scaling from the source code to the architectural level through the CSD. Theoretically, the CSD and its individual constructs are a separate issue from the automatic generation of the diagrams in a production environment. However, unless CSDs (or any other diagrams) can be automatically generated, they will not be utilized in practice.

6.2. New Features for GRASP

In further support of the notion of scalability discussed above, two additional features are planned for the GRASP software tool. First, the CSD window should provide the user with the capability to collapse the CSD based on all control constructs as well as complete diagram entities (e.g., procedures, functions, tasks and packages). This capability directly combines the ideas of chunking with control flow which are major aids to comprehension of software. Used in conjunction with the unit symbols, the elided CSD resulting from collapsing the diagram would provide a visualization of the software at a higher level of abstraction than the traditional CSD and would use a standard graphical notation to reference architectural level diagrams.

We also intend to extend GRASP to provide automatic generation of architectural level object diagrams. In addition to the CSD code-level visualization, GRASP will also produce object diagrams which support hyperlink navigation to and from the CSDs.

6.3. Evaluation

The current preference survey instrument should be refined with respect to the performance characteristics. The diagram types (FC, NS, WO, AD) that were compared to the CSD should be reassessed as to their extent of use in current practice. Non-graphical PDL or source code should also be considered as a treatment in any subsequent evaluation of preference.

While the analysis of preference data in this research clearly indicated statistically significant differences which heavily favored the CSD, a controlled experiment should be done to evaluate actual increases and/or decreases in comprehension due to the use of a particular graphical notation or PDL. In fact, since PDL is in such widespread use as a detailed design language, an experiment comparing the comprehensibility of PDL and the CSD would be perhaps even more appropriate than attempting to compare numerous graphical notations.

7. Conclusions

The control structure diagram is a graphical representation which maps directly to source code and PDL. The CSD offers advantages over previously available diagrams in that it combines simple intuitive graphical constructs with the best features of PDL and code without disrupting their familiar appearance. The potential of the CSD can be best realized during detailed design, implementation, verification and maintenance. The CSD can be used as a natural extension to popular architectural level representations such as data flow diagrams, object diagrams, and structure charts.

The GRASP project has provided a strong foundation for the automatic generation of graphical representations from existing software. The current prototype provides the capability for a user to generate the Control Structure Diagram (CSD) from source code in a reverse engineering mode with a level of flexibility suitable for practical application. The prototype is being used in three to five computer science courses per quarter at Auburn University, ranging from the ACM CS 1 course to a graduate level software engineering course. GRASP is used as the primary development environment for CS 1, CS 2, an undergraduate algorithms course, and an undergraduate software engineering course, and also as a reverse engineering tool and as a stepwise refinement tool to facilitate correctness verification in IBM's Cleanroom approach to software engineering (Linger, 1993) in a graduate software engineering course. The feedback provided by the students has been very useful, especially with respect to the user interface. The current GRASP prototype is freely available at the URL

`http://www.eng.auburn.edu/grasp`

An important issue for all software tools in general, and graphical representations in particular, is evaluation. An evaluation based on preference was conducted to provide information on user perceptions of the CSD. An experiment was designed and data was collected from software engineering students. Statistical analysis indicated highly significant differences among five graphical notations when compared with respect to eleven perfor-

mance characteristics. There was a clear preference for the CSD for seven of the eleven performance characteristics. Experience indicates that empirical evaluation of the comprehensibility (rather than preference) of graphical notations such as data flow diagrams, object diagrams, structure charts, and flowgraphs is difficult. However, such an evaluation for the CSD and GRASP tool is planned and will provide further insight into the role that graphical notations play in the comprehension of software and, as a result, their potential impact on the overall cost of software.

The primary impact of reverse engineering graphical representations will be improved comprehension of software in the form of visual verification and validation (V&V). Graphically-oriented editors must provide capabilities for dynamic reconstruction of the diagrams as changes are made to other diagrams at various levels. These graphical representations should provide immediate visual feedback to the user in an incremental fashion as individual structural and control constructs are completed. The CSD becomes a natural detailed-level graphical extension for system and architectural level diagrams. In this capacity, the CSD has the potential to replace traditional non-graphical PDL used in software design as well as plain text source code used in implementation, testing, and maintenance. The current prototype of the CSD generator, while only one of a set of required visualization tools, has clearly indicated the utility of the CSD. Future enhancements will only increase its effectiveness as a tool for improving the comprehensibility of software

Acknowledgments

This research was supported, in part, by a grant from Marshall Space Flight Center, MSFC, AL 35812. The assistance provided by NASA personnel, especially Mr. Robert Stevens and Mr. Keith Shackelford, has been extremely beneficial. Several graduate students were instrumental in developing the prototype software, including Larry Barowski, Karl Mathias, and Joseph Teate.

Notes

1. The 33 students left some responses blank. Averages were computed by dividing by the actual number of responses shown in Table 1.

References

Aoyama, M. 1989. Design specification in japan: Tree-structured charts. *IEEE Software*: 31–37.

Baecker, R. M., DiGiano, C., and Marcus, A. 1997. Software visualization for debugging. *Communications of the ACM* 40(4): 44–54.

Baecker, R. M., and Marcus, A. 1990. *Human Factors and Typography for More Readable Programs*. ACM Press.

Barnes, J. G. P. 1984. *Programming in Ada*. Menlo Park, CA: Addison-Wesley, 2 edition.

Booch, G., and Bryan, D. 1994. *Software Engineering with Ada*. Benjamin/Cummings, third edition edition.

Chikofsky, E., and Cross, J. 1990. Reverse engineering and design recovery—a taxonomy. *IEEE Software*: 13–17.

Conover, W. J. 1980. *Practical Nonparametric Statistics*. New York: John Wiley and Sons.

Cross, J., Chikofski, E. J., and May, C. H. 1992. Reverse engineering. *Advances in Computers* 35: 199–254.

Cross, J., and Sheppard, S. V. 1988. The control structure diagram: An automated graphical representation for software. *Proceedings of the 21st Hawaii International Conference on Systems Sciences*, vol. 2, IEEE Computer Society Press, 446–454.

Cross, J., Sheppard, S. V., and Carlisle, W. H. 1990. Control structure diagrams for ada. *Journal of Pascal, Ada, and Modula 2* 9(5).

Cross, J. H. 1994. Improving comprehensibility of ada with control structure diagrams. *Proceedings of the Software Technology Conference*, distributed on CD–ROM, 25 pages, April 11-14, Salt Lake City, Utah.

Cross, J. H., Chang, K. H., and Hendrix, T. D. 1996. Grasp/ada95: Visualization with control structure diagrams. *CrossTalk Journal of Defense Software Engineering* 9(1): 27–45.

Cross, J. H., and Hendrix, T. D. 1996. *Software Visualization*, volume 7 of *Series on Software Engineering and Knowledge Engineering*, chapter Language Independent Program Visualization, 27–45. World Scientific.

Curtis, B., Sheppard, S., Kruesi-Bailey, E., Bailey, J., and Boehm-Davis, D. 1989. Experimental evaluation of software documentation formats. *Journal of Systems and Software* 9: 167–207.

Green, T., and Petre, M. 1992. When visual programs are harder to read than textual programs. *Proceedings of the Sixth European Conference on Cognitive Ergonomics (ECCE-6)*, Budapest, Hungary.

Green, T., Petre, M., and Bellamy, R. 1991. Comprehensibility of visual and textual programs: A test of superlativism against the match-mismatch conjecture. *Empirical Studies of Programmers Fourth Workshop*. Ablex.

Kissel, G. 1995. Effect of computer experience on subjective and objective software usability measures. *Proceedings of the Conference on Human Factors in Computing Systems*, 284–295.

Linger, R. 1993. Cleanroom software engineering for zero-defect software. *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, MD, 2–13.

Marshall, T. E., Byrd, T. A., Gardiner, L. R., and Rainer, R. K. 1998. Technology acceptance and performance: An investigation into requisite knowledge (forthcoming). *Information Resources Management Journal*.

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

McQuaid, P. A., Chang, K. H., and Cross, J. H. 1995. A complexity metric to aid software testing and maintenance. *Proceedings of the Decision Sciences Institute* 2.

Moher, T., Mak, D., Blumenthal, B., and Leventhal, L. 1993. Comparing the comprehensibility of textual and graphical programs: The case of petri nets. *Empirical Studies of Programmers: Fifth Workshop*, 137–161.

Petre, M. 1995. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM* 38(6): 33–44.

Price, B. A., Baecker, R. M., and Small, I. S. 1993. A principled taxonomy of software visualization. *Journal of Visual Languages and Computing* 4(3): 211–266.

Raeder, G. 1985. A survey of current graphical programming techniques. *IEEE Computer* 18(8): 11–25.

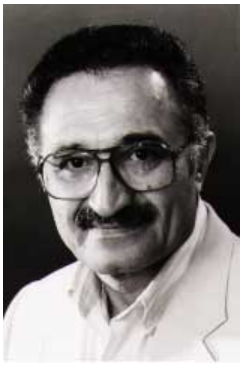
Scanlan, D. A. 1989. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*: 28–36.

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

Shu, N. C. 1988. *Visual Programming*. New York, New York: Van Norstrand Reinhold, Inc.

Tripp, L. L. 1989. A survey of graphical notations for program design. *ACM Software Engineering Notes* 13(4): 39–44.

Dr. Cross is professor and chair of Computer Science and Engineering at Auburn University. His primary interests are teaching undergraduate and graduate courses in software engineering and directing research in the areas of software methodology, testing, and reverse engineering. His continuing research efforts include the GRASP project which focuses on reverse engineering and the automatic generation of graphical representations of software. The purpose of the generated visualizations is to improve the comprehensibility of software, and as a result, increase productivity and reliability across all activities that involve code reading. Dr. Cross is a member of ACM and a senior member of IEEE Computer Society of which he is currently serving on the Board of Governors.



Dr. Maghsoodloo is a professor of Industrial and Systems Engineering at Auburn University. His research interests and publications are in nonparametric statistics, multi-variate analysis, experimental design and Taguchi methods, regression correlation / time-series analysis, quality control, and reliability engineering. In addition to being an associate editor of IIE Transations and a registered professional engineer, he is a member of ASA and ASQC.

Dr. Hendrix is an assistant professor of Computer Science and Engineering at Auburn University. His research interests include software methodology, software metrics, reverse engineering, software visualization, and programming languages. Currently, Dr. Hendrix is focused on applying software visualization techniques to address issues relating to industrial software engineering. The primary goal of his continued research in this area, including his work with the GRASP project, is to effect fundamental improvements in software engineering best practices. Dr. Hendrix's teaching interests are undergraduate and graduate courses in software engineering and database systems. Dr. Hendrix is a member of ACM and IEEE Computer Society.