Empirical Software Engineering, 2, 361–379 (1997)
© 1997 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Visual Depiction of Decision Statements: What is Best for Programmers and Non-Programmers?

JAMES D. KIPER
Department of Systems Analysis, Miami University, Oxford, OH 45056

BRENT AUERNHEIMER
Department of Computer Science, California State University, Fresno, CA 93740

CHARLES K. AMES
Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA 91109

Abstract. This paper reports the results of two experiments investigating differences in comprehensibility of textual and graphical notations for representing decision statements. The first experiment was a replication of a prior experiment that found textual notations to be better than particular graphical notations. After replicating this study, two other hypotheses were investigated in a second experiment. Our first claim is that graphics may be better for technical, non-programmers than they are for programmers because of the great amount of experience that programmers have with textual notations in programming languages. The second is that modifications to graphical forms may improve their usefulness. The results support both of these hypotheses.

Keywords: visual programming, decision structures, program comprehension, expert-novice differences

1. Introduction

The increase in sophisticated graphical user interfaces for diverse systems can be attributed to maturing graphics software as well as affordable, powerful graphics hardware. There is corresponding activity in the application of graphical approaches to programming: in programming visualization, visual databases, languages for image processing, visual environments for textual languages, and visual programming languages (Burnett and Baker, 1993). The search for a truly visual programming language (Kiper et al., 1996) with experimentally verifiable benefits to user comprehension has been less than successful (Moher et al., 1993; Pandey and Burnett, 1993; Petre and Green, 1993). However, intuition suggests that appropriate graphics may have positive effects on some aspects of program comprehension.

This paper describes an investigation of the ability of programmers and non-programmers to comprehend visual representations of certain aspects of programming languages versus their ability to comprehend corresponding textual representations. This investigation begins with a replication of a prior experiment (Green et al., 1991), then extends that study with a second experiment.

2. Background

2.1. Visual Programming

Visual programming is sometimes touted as one method to increase programmer productivity. LabVIEW® (Vose and Williams, 1986) and Prograph® (Cox and Pietrzykowski, 1988) are two commercially available visual programming environments. These languages use icons and arcs to represent programs.[1] Other visual programming languages have been developed by researchers (Glinert, 1990; Chang, 1990; Ichikawa and Hirakawa, 1990; Meyers, 1990; Price et al., 1993), many for specific application areas. The quest for a general purpose visual programming language (Kiper et al., 1995) has been a goal for others.

It is a responsibility of the field of software engineering to validate the utility of new tools and methods, not just to propose them. Basili, Selby, and Hutchens (1986) describe scientifically sound methods of experimentation in software engineering. Fenton, Pfleeger, and Glass (1994) recently appealed for scientific investigation of software engineering claims.

The next two subsections describe previous experiments studying the use of visual representations of programs. This is not an exhaustive list, but a set that is pertinent to our experiments described in subsequent sections. First, we describe the experiment that was replicated.

2.2. Replicated Study

Our work was motivated by experiments (Green et al., 1991) that compared textual languages and the visual programming language LabVIEW. These experiments used programmers as subjects to study the comprehensibility of decision trees in textual and graphical forms. The conclusions of this work are similar to ours, finding that a textual form generally was better than a graphical form for programmer's comprehension of decision statements.

Green's experiment compared two graphical representations of decision statements to two textual representations. This study is distinguished from the majority of work in this area in that the graphical forms used were based on a data flow paradigm rather than control flow, and that the graphical forms examined were those of a commercially available language—LabVIEW®.

The experiment used a within subject, randomized design in which a sequence of three tasks addressed several issues. The hypothesis of interest to us is that there is significant difference between the results of the graphics and text trials. The experimenters made no specific predictions about which would perform better. They also explored the difference between "forward" and "backward" questions, and "sequential" and "circumstantial" language constructs. These two aspects tie this work to that of Green (1977) and Curtis et al. (1989).

A *forward question* is one that gives the user a particular true and false instantiation of input variables, and asks the user for the output condition produced by those inputs. A *backward question* asserts the truth of a particular output variables and requires the subject to determine settings of input variables that would produce this output.

```
if leafy :
   if green :  grill
   not green :
      if hairy :   roast
      not hairy :  boil
      end hairy
   end green
not leafy :
   if cold :   boil
   not cold :  fry
   end cold
end leafy
```

Figure 1. Sample of nested if text form.

A *sequential language* is one that allows the user to move easily from input conditions to outcomes; a *circumstantial language* allows the use of a conclusion or outcome to determine conditions that produced that outcome. Some previous work had shown that, for experienced programmers, sequential programs were more appropriate for forward questions; consequential programs were better when the task was to answer backward questions (Davies, 1989).

The two graphic forms and two textual forms used were used in our replication. They are described in detail here. One textual form was based on the "Nest-INE" notation developed by Sime et al. (1977). As can be seen in Figure 1, this form is comprised of a sequence of nested "if-then" structures in which indentation used to indicate the depth of nesting. This form was classified as *sequential*, and we will refer it as the "Nested If". The second textual form, illustrated in Figure 2, is reminiscent of Prolog syntax since the conclusion of the decision is given first followed by an if condition. When the antecedent is true, the conclusion is asserted giving a *circumstantial* construct. This form uses symbolism for logical connectives: '&' for and, '|' for or, '¬' for not, and will be referred to as the "Do If" form.

The two graphic forms from LabVIEW will be referred to as the "Gates" and "Boxes" notation. (See Figures 3 and 4.) The "gates" form presents decision statements as a wiring diagram (Figure 3). Antecedents are listed in a sequence of boxes along the left side of the diagram. Possible results (conclusions) are listed in boxes on the right side. These are connected using engineering symbols for AND, OR, and NOT gates. Green

Figure 2. Sample of do if text form.

and his colleagues categorized these gates as *sequential* for the natural flow from inputs to conclusions.

The boxes form is a series of nested boxes each of which represents a particular true or false setting of input variables. The veracity of result variables is represented by highlighting of the 'T' and 'F' in the associated boxes on the right side of the diagram. (When a condition is true, the 'T' in its box is highlighted and the 'F' is dimmed; otherwise the 'F' is highlighted and the 'T' is dimmed.) When the user clicks on the true/false switch, the setting is toggled. By toggling these switched to various settings, a user can explore the entire decision structure (Figure 4). The gates form was categorized as a *circumstantial* form since it is relatively easy to see what inputs lead to a particular outcome.

In addition to presenting a decision statement, each stimulus screen presents a question about this statement to be answered by the subject. These questions were used in two forms—forward and backward. In all cases there was a unique correct answer. The logical "or" operator in a decision condition allowed for two settings to produce the correct answer.

Subjects in this study were five domain experts who each had at least six months of experience in the use of LabVIEW. The small number of subjects is one of the reasons that a replication was important.

The independent variables in this experiment were form modality (text/graphics), structure (sequential/circumstantial), and direction of question (forward or backward). The dependent variable was response time which was used to quantify "comprehension." Thus, there were eight combinations of treatments. All subjects were presented with 16 questions representing two of each of these eight combinations of treatments. Two of each were required to allow for questions without an "or" that had one answer (single questions), and questions with an "or" that had two answers (double question). Subjects were asked to find both answers in the cases that an "or" was used. These were presented in a balanced, randomized design.

Thus, for forward questions subjects were able to answer with a single button click that represented the outcome of the decision statement. Backward questions required subjects to indicate the settings of all input variables to the decision statement that would cause

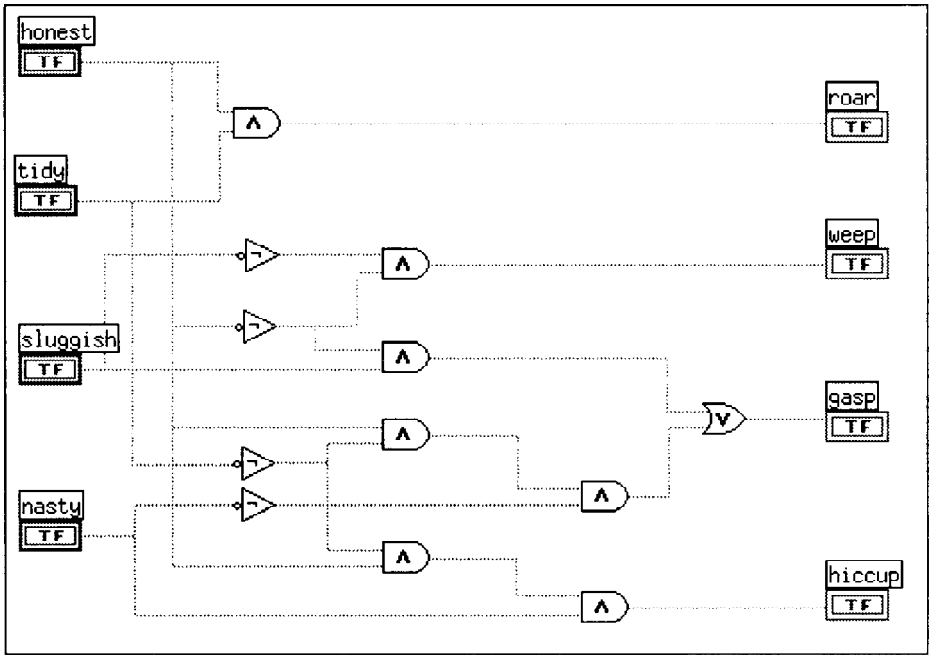


Figure 3. Sample of gates graphic form.

a given output. Thus, backward, single questions required six button clicks; backward, double questions required 12 clicks. The appendix contains sample screens.

The experiment was conducted on a Macintosh computer with a 1152×870 pixel, black-and-white monitor. The stimuli were presented using a program written in SuperCard 1.5. Portions of the screen were made sensitive to mouse clicks to simulate the behavior LabVIEW boxes decision statements. All responses were by clicking buttons with a mouse. An untimed practice session of eight screens allowed user to experience all eight treatment combinations before beginning the timed portion. Subjects clicked a "Done" button when they were satisfied with their responses. After clicking "Done" they were presented with a blank screen with a single "Next" button. Clicking the "Next" button revealed the next screen. They were allowed to rest at this screen before proceeding to the next screen. The SuperCard program timed them from the time they click "Next" until they click the "Done" button. Subjects were given no feedback about the accuracy of their answers or their time to complete.

Green et al. analyzed time scores of all responses including erroneous ones, which were few. Separate analyses of variance (ANOVAs) were performed for forward questions and backward questions. These were not combined since each forward question required only one mouse click to response, where backward questions required 6 clicks (for single path)

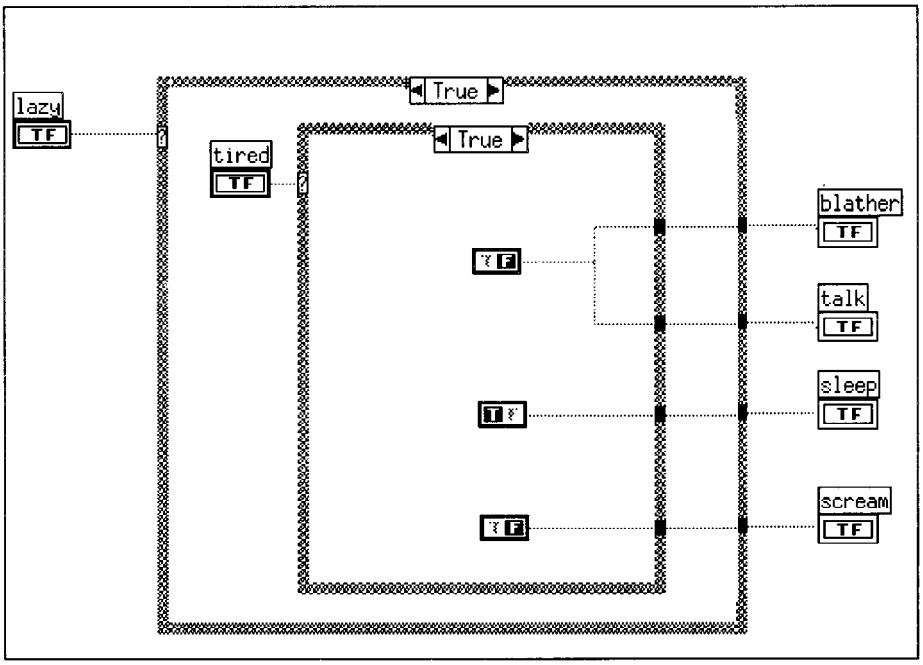


Figure 4. Sample of boxes graphic form.

and 12 clicks (for double path) questions. For forward questions, the two responses for each treatment were averaged. For backwards questions, response timings were combined with a weighted average: $(2 * \text{single-path} + \text{double-path})/3$. The ANOVAs showed one strongly significant main effect of Text versus Graphics with $F(1, 4) = 52.13$, $p = 0.002$. There was another significant interaction, Direction versus Structure with $F(1, 4) = 11.41$ and $p = 0.028$. Figures 5 and 6 illustrate these analyses. Thus, the hypothesis that there is a difference between text and graphics is strongly supported, and, in particular, text response times were significantly less than graphic response times.

2.3. Prior Studies

One of the classic experiments in this area examined the utility of detailed flowcharts (Shneiderman et al., 1977). In this group of five experiments, Shneiderman and his colleagues studied the utility of detailed flowcharts in composing, comprehending, debugging, and modifying programs. They found no statistically significant differences between the performance of subjects who used flowcharts to those who did not. Scanlan (1989) reached different conclusions about the utility of structured flowcharts to aid program comprehension. He found a statistically significant advantage in using flowcharts to aid program

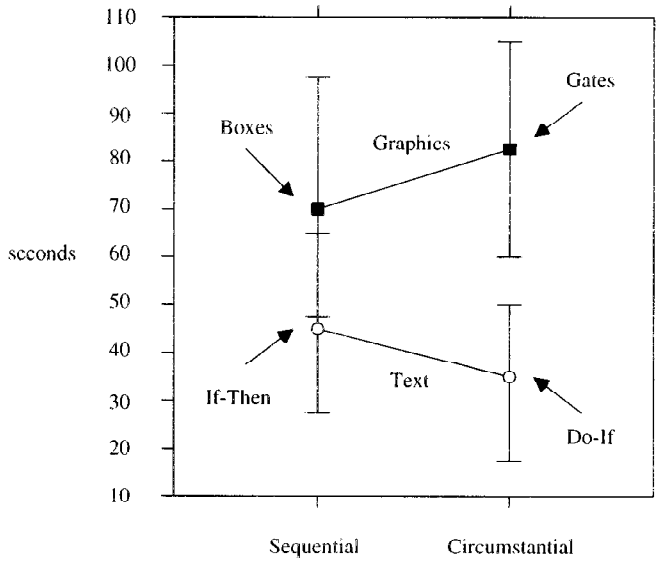


Figure 5. Modality by structure: response times from Green et al. (1992) (cell means with 95% confidence bars).

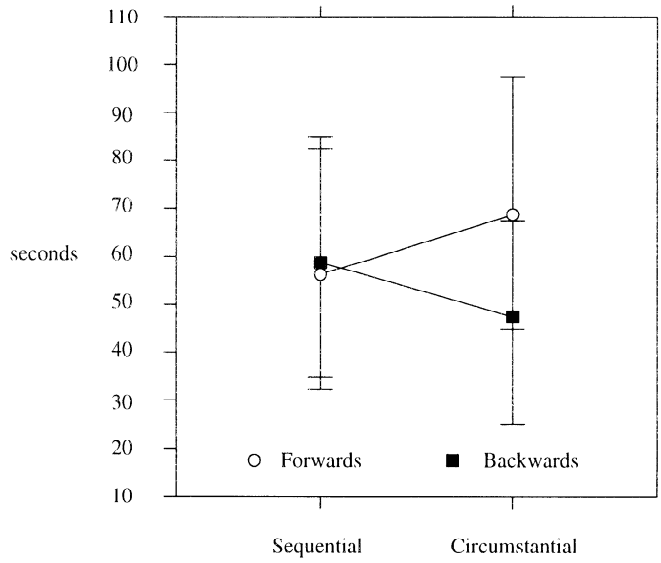


Figure 6. Question direction versus structure: response times from Green et al. (1992) (cell means with 95% confidence bars).

comprehension. He found a statistically significant advantage in using flowcharts versus pseudocode for comprehension of complicated algorithms. Scanlan also measured comprehension by response time.

The experimental work of Green and colleagues described above was extended to Petri nets (Moher et al., 1993) with analogous results. Petre (1995) provides an excellent summary of experimental results in this area. Pandey and Burnett (1993) studied constructability of programs in visual versus textual languages in the domain of matrix manipulation, concluding that construction was easier in a visual language.

Von Mayrhauser and Vans (1995) provide an extensive table of published observational studies, correlational studies, and experiments examining various aspects of program comprehension. These studies generally examine textual programming languages. However, they can serve as a baseline for comparison to visual languages, and a source for understanding of cognitive processes involved in program comprehension.

3. Investigation Goals

The goals of our investigation are threefold. First, we replicated the study of Green et al. (1991). Thus, we based our experimental design on theirs. The second goal was to determine if graphics are more easily comprehended than text for technical, non-programmers. This required a second experiment with alternate subjects whose performance was compared to that of the programmers in the first study. A third goal was to determine if some adjustments to the graphical forms used could increase their comprehensibility. Thus the hypotheses that were studied are:

- Null Hypothesis #1: Comprehension of text and graphic forms of decision statements as measured by response time are equal.
- Null Hypothesis #2: Graphic rather than textual representations of decision statements have no effect on the comprehension of decision trees for technical, non-programmers compared to that of programmers.
- Null Hypothesis #3: An appropriate modification in graphical form that makes components of that form easier to see does not change its comprehensibility.

In the following section, we describe the replication that addresses hypothesis 1. Then we describe the second experiment and hypotheses 2 and 3. First, let us motivate why it was interesting to us to consider the use of graphical (visual) programming constructs with *technical, non-programmers*.

This study distinguishes between programmers and others who are technically apt, but are not programmers. This distinction is based on the working hypothesis that these two groups view their tasks quite differently, even though these tasks frequently overlap and require inter-group technical communication and consultation.

Programmers in this experiment are experienced software developers. Members of this group generally view solutions to problems as procedural algorithms. *Technical, non-programmers* regularly work with technology, but do not regularly develop or maintain

programs. They may have had an introductory class in programming and have written small programs, but this is not a part of their daily work. This group, however, excludes users whose work does not involve the use or development of technology. That is, *technical, nonprogrammers* incorporates engineers, support staff for engineers, and technical managers. Another categorization of these users is that they are the experts in technical application areas from whom programmers obtain and validate requirements for new or modified software systems.

Although the technical, non-programmers do not create or modify software as a part of their work, they are application experts. As such, they often interact with software developers. Their involvement emphasizes requirements gathering and analysis, although their participation during design and implementation can be beneficial. Their active involvement in code or design walkthroughs is sometimes limited by lack of knowledge of programming or design languages. Visual representations of code or designs may aid comprehension more for this user class than for programmers.

4. Empirical Studies

4.1. Replication

The subjects in this experiment were all programmers with more than three years of programming experience. None had more than cursory experience with LabVIEW programming. The experimental method used was the same as that of the experiment of Green et al. (1991). The 16 screens used as stimuli were identical to those used in that previous experiment. We were unable to obtain a working version of the original SuperCard driving program, but were able to reconstruct it from pieces.

The primary differences from this experiment and the study of Grenn et al. were these:

1. Our replication involved nine subjects versus five of the original experiment.
2. Our programmers were experience programmers, but were not experienced in the use of the graphical programming language LabVIEW. The subjects of the prior experiment had at least six months of experience with LabVIEW and five to fifteen years of general programming language experience.
3. Our experiment was conducted on a PowerMac 840AV with a 1024 by 768 pixel color monitor. Their experimental apparatus was a Macintosh II with a 1152 by 870 pixel black and white monitor.

4.2. Subsequent Experiment

Hypothesis 2 required a two factor experiment: programmer versus non-programmer subject types, and graphic versus textual form of decision statements. Since the replication experiment described above included programmers using text and graphical forms, additional technical, non-programmers subjects were needed. Conclusions could then be drawn by comparison of data from these two experiments.

In this experiment we again used the two text and two graphical forms of decision statements described previously. Each of these four forms is used to represent decision trees that give possible outputs (conclusions) for various settings of Boolean, input conditions (antecedents). The two text forms and the two visual forms allowed the evaluation of multiple forms with each subject in a balanced manner to address any bias the subject has toward one form or the other.

To investigate Hypothesis 3, the *gates* notation was modified to allow users to click on an output condition. The software responded by color highlighting the wires that led into that boxes. A similar highlighting occurred when users clicked on input conditions. This modification seemed reasonable since subjects in the first experiment complained that the lines in the gates form were difficult to follow, especially when these lines crossed.

This modified version of the experiment used both programmers and technical non-programmer as subjects. It also involved the modified gates graphical form. Comparison of the response times of programmers and non-programmers in the experiment gave us data pertinent to Hypothesis 2. A comparison of the response times of programmers on the modified gates notation to that of programmers on the original gates notation in our first experiment gave data pertinent to Hypothesis 3.

4.3. Experimental Design and Method

The experimental design of the replication experiment was identical to that of the experiment of Green et al. (1991). That is, it was a within subject, randomized design. To obtain data that is relevant to Hypotheses 2 and 3 it was necessary to adapt this experiment slightly. The experimental design used to address Hypothesis 2 required a between subject study in which data from programmers was compared to that from non-programmers. For Hypothesis 3, the experiment was a between subject design comparing performance of programmers in the first experiment with the original *gates* notation to that of another group of programmers in the second experiment using the modified *gates* notation. In all cases, the randomized design presented 16 stimulus screens to all subjects. As described previously, response time were used to measure comprehensibility.

The experimental method for the replication experiment was identical to that of Green and his colleagues (1991). The independent variables in this experiment were form modality (text/graphics) and subject type (programmer/technical non-programmer). The dependent variable was response time which was used to quantify "comprehension." The method used to conduct the second experiment was identical with the exception of the use of the modified *gates* notation and the fact that both programmers and non-programmers were used. We repeat the description of this method here. The experimental apparatus consisted of a Power Macintosh and software written in SuperCard. Each screen presented to the subject one of four representations of a unique decision tree—two graphic forms and two textual forms. Each of the two graphic and two text forms were used in four different screens, for a total of 16 treatments for each subject. The order of screen presentation was randomized between subjects. Each of these stimuli screens was preceded and followed by a dark screen with a button labeled "Next." The software determined the length of time between the presentation of each stimulus screen and the subjects' indication that they were finished with that screen.

The software includes a set of screens in a pre-trial learning phase that presents two of each of the four forms before the timed screens are presented. These initial screens were used as a learning phase to introduce subjects to the environment and to each of the four decision statement forms. The experimenter was present during this pretrial phase to give brief instructions and to answer questions. When subjects finished these initial screens and were comfortable with the environment they proceeded to the trials. An experimenter was present during the entire session to give quick answers when subjects were had problems.

5. Data Analysis

5.1. Replication

For this replication, we used the same analyses that Green and colleagues (1991) used for their experiment. Our original subject pool for this experiment consisted of 10 programmers. One subject had an extraordinary amount of trouble with some of the stimulus screens—particularly the graphical ones. (His average response time on the gates notation was more than 7 standard deviations above the mean; for the boxes notation, his response times were more than 9 standard deviations above the mean. His response times for the textual notations were above the mean, but not as extreme.) Although we believe that we would be justified in excluding data that is this far from the mean, we have included data analysis for data including this outlier and again for data excluding this.

The same analysis of variance statistical procedures were used to examine the resultant response times as were used in the prior experiment. This procedure used a weighted average for the response time of backwards questions since some question types ("double path") required twice as many response clicks as others ("single path"). First, we give the data analysis for data excluding the outlier. The ANOVA showed a strong significance to the effect of Text versus Graphics with $F(1, 16) = 15.87$ and $p < 0.01$. We used this form of analysis since this was the type used in the replicated experiment. Since it is not obvious to us that the samples have the same variances, we also used a t test assuming unequal variances in our analysis. The differences in means were significant at the 0.01 level with a pooled variance of 3.5 and $t = 3.98$. (The critical value of t is 3.106.)

An analysis of all the data show a significance, albeit not as strong as the previous, to the effect of Text versus Graphics with $F(1, 18) = 5.46$ and $p < 0.05$. The differences in means were significant at the 0.05 level with a pooled variance of 3.5 and $t = 2.33$. (The critical value of t is 2.26.)

Table 1 gives the means and standard deviations for the four notations. Figure 7 presents 95% confidence these results in the same format as does Figure 5 for the original experiment. In particular, the data of Table 1 and the ANOVA for data considered with or without the outlying data *supports the rejection of null Hypothesis 1*—and thus replicates the results of the original experiment.

Table 1. Means and standard deviations from replication experiment.

Outlier Excluded		
Notation Type	**mean**	**standard deviation**
gates	193.0 sec.	82.3
boxes	106.9 sec.	51.4
if-then	66.8 sec.	37.4
do-if	57.24 sec.	19.5
All Data		
Notation Type	**mean**	**standard deviation**
gates	251.5 sec.	200.5
boxes	156.9 sec.	165.5
if-then	75.1 sec.	43.8
do-if	59.8 sec.	20.0

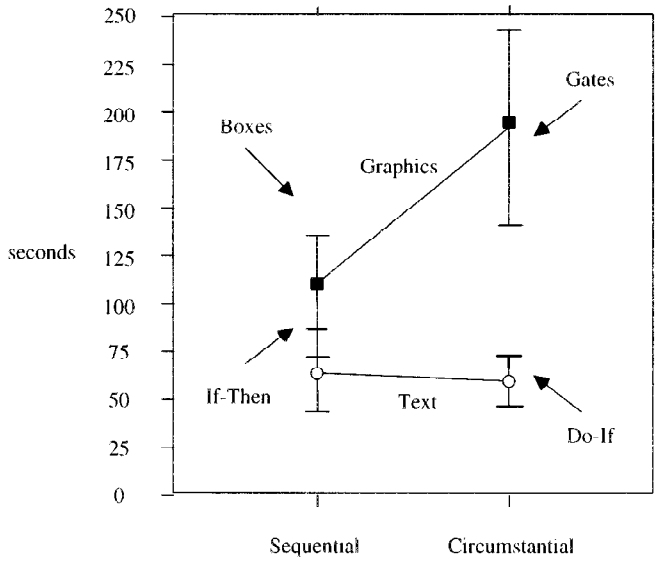


Figure 7. Replication experiment—modality by structure: response times (cell means with 95% confidence bars).

Table 2. Programmer versus non-programmer statistics.

subject category	number	mean	standard deviation
programmer	9	128%	0.572
non-programmer	8	77.76%	0.308

Table 3. Original gate notation versus modified gate notation.

subject category	number	mean	standard deviation
Original gate notation	9	193.0 sec.	82.3
modified gate notation	9	92.2 sec.	30.8

5.2. Extension

Null Hypothesis 2 is that graphic rather than textual representations of decision statements have no affect on the comprehension of decision trees for technical, non-programmers compared to that of programmers. The second experiment described previously collected data from both programmers and non-programmers on both text and graphic notations. The statistic used was the percentage of change from text to graphic. We calculated this as (average graphic response time − average text response time)/average text response time. Note that for both programmers and non-programmers, text response times were less than graphic response times (Table 2). Thus a smaller percentage of change meant that the graphic notation response time increased less. An ANOVA showed that the difference was indeed significant at the 0.05 level ($F(1, 15) = 4.9389$ and $p = 0.0421$). The t test gave a pooled variance of 3.5 and a t value of 2.300 which exceeds the critical value ($\alpha = 0.05$) of 2.179 *allowing us to reject null Hypothesis 2*. We conclude that the difference between programmers and non-programmers is significant. The means shows that this difference favors the non-programmers.

Null Hypothesis 3 is that an adjustment in graphical form does not change its comprehensibility. To obtain data relative to this hypothesis we examined the performance of programmers from the first experiment on the original *gates* notation to that of programmers in the second experiment on the modified *gates* notation. Note that only data for programmers from the second experiment were used to make this a fair comparison. Table 3 reports the relevant statistics for comparison of these two graphical notations. An ANOVA showed that the difference was significant at the 0.001 level ($F(1, 16) = 11.86$ and $p = 0.00334$.) The t test assuming unequal variances gave similar results. The t value was 3.44 which exceeds the $\alpha = 0.01$ p critical value of 3.17.

Despite that fact that a distinct set of programmers was used for the second experiment from the first avoiding any learning effect, and the fact that the *boxes*, *if-then*, and *do-if* stimulus screens were identical in the two experiments, there was a general improvement in programmer subject performance, i.e., lower response times, in the second experiment. We

Table 4. Original gate notation versus modified gate notation with adjustment.

subject category	number	mean	standard deviation
Original gate notation	9	193.03 sec.	82.25
Modified gate notation (adjusted)	9	104.22 sec.	34.79

increased the modified *gates* responses times by this 13% general improvement to adjust for this general improvement. The difference between the mean of response times for *gates* and mean for response times of *modified gates* with this adjustment was still significant with $F(1, 16) = 8.90$ and $p = 0.00878$. The t test assuming unequal variances gives a pooled variance of 3.5 and a t value of 2.983. Since the 0.01 critical value is 2.764, the difference in means is significant at the 0.01 level. Table 4 reports the means and standard deviations with this adjustment. Thus, the statistical analysis of the data in Tables 3 and 4 *allows us to reject null Hypothesis 3*.

5.3. Threats to Validity

In terms of internal validity, these experiments are based on the commonly used assumption that response times are a valid measure of comprehensibility. This has been an accepted measure of comprehensibility of programming notations in many prior experiments including those of Shneiderman et al. (1977), Scalan (1989), Green, Petre, and Bellamy (1991). Thus, its validity is accepted here. The validity of our replication may be questioned because our subjects lack of experience with LabVIEW programming, where subject in Green's experiment had at least six months' experience. Thus, our programmer subjects may have been more biased toward the textual forms used. However, the fact that non-programmers also comprehended the textual forms more quickly than the LabVIEW visual forms gives more validity to the experiments. The fact that color monitors were used in our experiments where black and white were used in the replicated experiment was not viewed as significant since color was not used to give visual cues except by highlighting. Highlighting can also be accomplished with reverse video on black and white monitors. Another threat to the internal validity is that our two experiments were carried out several weeks apart. However, this is not significant since different subjects were used, and the comparison of response times for the two styles of *gates* were adjusted for overall differences in response times for all notations.

The primary threat to the external validity of this experiment is that subjects' response times were measured in a laboratory environment rather than in a typical work environment. This is a typical problem necessitated by the need to control other aspects of the environment. The general applicability to visual programming languages is limited by the fact that

this experiment used decision statements from a data flow programming languages. The implications of this research to functional or more typical imperative visual languages and to other control structure, e.g. loops, is limited by this.

Another threat to the external validity of these experiments is that the tasks involved are very simple. Comprehension of decision statements is only a small portion of the complex process of software development.

6. Conclusions

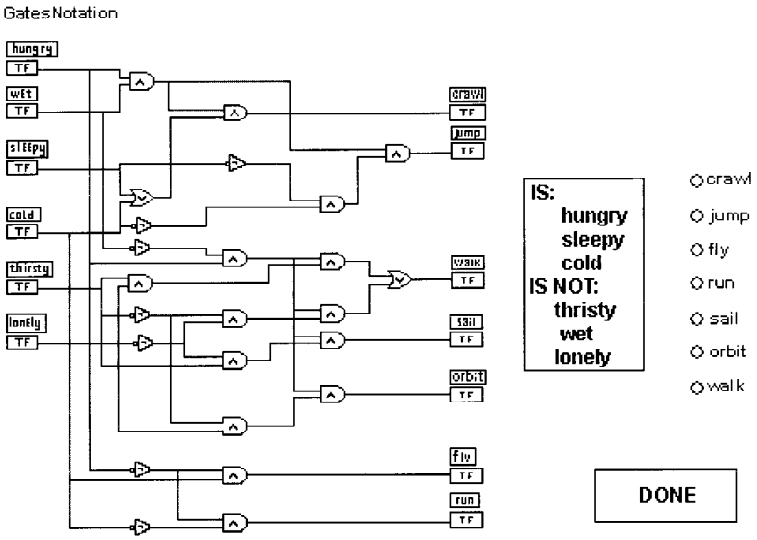
Our analysis of the data from our experiments has clearly allowed us to reject the null Hypothesis 1 which replicates the conclusion of the study of Green et al. (1991). Specifically, this is the conclusion that textual notation for decision statements are easier to comprehend for both programmers and technical, non-programmers than are the visual representations of decision statements used in the commercially available programming language LabVIEW.

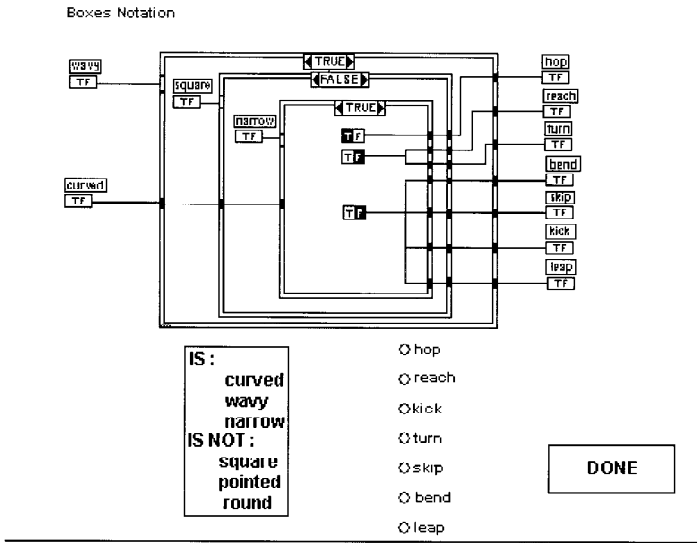
The data analyses of the experiment data allow us to reject the second null hypothesis. From the data, it was clear that, for technical, non-programmers, the difference between comprehension times for graphic and textual form were less than the difference for programmers. Although this experiment does not reveal the source of this variation, we suspect that it is produced in large measure by the extensive experience that most programmers have with textual programming languages.

The data analysis also supports the alternate hypothesis over the third null hypothesis. To summarize, this hypothesis is that appropriate changes to a graphic form can have a positive effect on the comprehensibility of this form. In our experiments, this change from the first to the second experiment was motivated by feedback from subjects about the difficulty in using one of the graphic notations, specifically, the gates form. This is not a surprising conclusion. Some graphic forms are easier to comprehend than others. However, we found this conclusion to be a useful validation of our intuition.

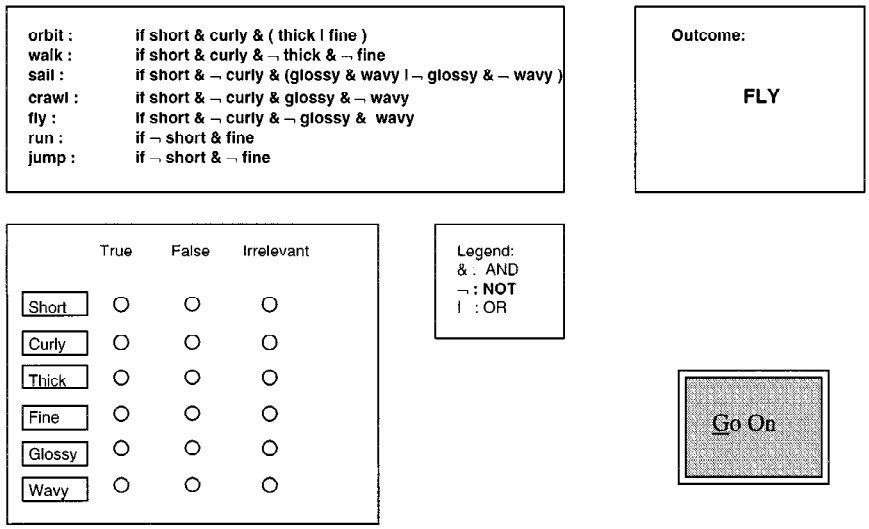
The value and power of visual forms of representations in other fields leads us to the intuition that there may be some task and audience for which some visual programming notations are useful. It seems that the great amount of experience that programmers have had with textual representations of programming languages will obscure any beneficial effects of visual representations for comprehension. These experiments have studied only a small, simple comprehension activity that is part of a much more complex software development process. Despite the simplicity of the experimental tasks, the use of conditionals is a fundamental building block of software development. However, we have not yet studied the effect of visual notations on program recall, or on comprehension or recall of other syntactic constructs, e.g. loops. These remain for future work.

Appendix: Sample Stimulus Screens

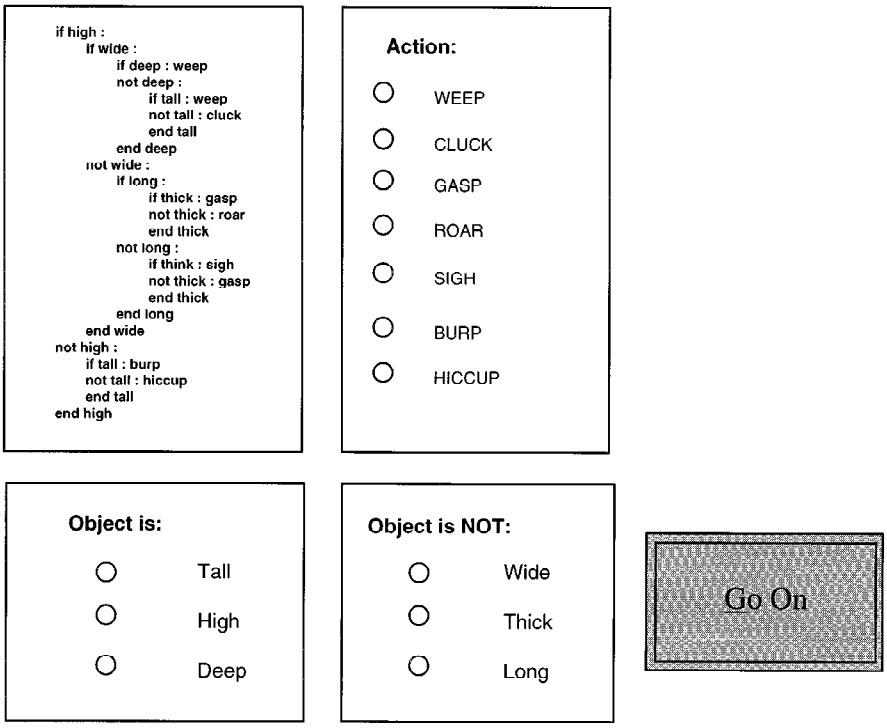




Do-If Notation



Nested-IF Notation



Acknowledgments

The work described in this paper was performed at the Jet Propulsion Laboratory, California Institute of Technology, under a construct with the National Aeronatics and Space Administration. Dr. Kiper and Dr. Auernheimer were supported by a NASA/ASEE Summer Faculty Research Fellowship. Dr. Kiper was additionally supported by a sabbatical provided by Miami University.

The authors thank their JPL colleagues and subjects for help with this study. In particular, without Dave Hermsen's support at JPL this work would not be possible.

Notes

1. Languages such as VisualBasics® and Visual C++® are useful for development of graphical user interfaces, but still require textual programming.

References

Basili, V. R. A., Selby, R. W., and Hutchens, D. H. 1986. Experimentation in software engineering. *IEEE Transactions on Software Engineering* 12(6): 758–773.

Burnett, M. M., and Baker, M. J. 1993. A classification system for visual programming languages. Technical report 93-60-14. Oregon State University, June 1993.

Chang, S., ed. 1990. *Visual Languages and Visual Programming*. New York: Plenum Press.

Cox, P. T., and Pietrzykowski. 1988. Using a pictorial representation to combine dataflow and object-orientation in a language independent programming mechanism. *Proceedings International Computer Science Conference*, 695–704.

Curtis, B., Sheppard, S., Kruesi-Bailey, J., and Boehm-Davis, D. 1989. Experimental evaluation of software documentation formats. *Journal of Systems and Software* 9(2): 167–207.

Davies, S. P. 1989. Skill levels and strategic differences in plan comprehension and implementation in programming. In A. Sutcliffe and L. Macaulay (eds.), *People and Computers V*. Cambridge University Press.

Fenton, N., Pfleeger, S. L., and Glass, R. L. 1994. Science and substance: a challenge to software engineers. *IEEE Software* 11(4): 86–95.

Glinert, E. P., ed. *Visual Programming Environments: Applications and Issues*. Los Alamitos, CA: IEEE Computer Society Press.

Green, T. R. G. 1977. Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology* 59: 93–109.

Green, T. R. G., Petre, M., and Bellamy, R. K. E. 1991. Comprehensibility of visual and textual programs: a test of superlativism against the 'match-mismatch' conjecture. Empirical studies of programmers: fourth workshop, Jürgen Koenemann-Belliveau, Thomas G. Moher, Scott P. Robertson (eds.). Norwood, N.J.: Ablex Pub. Corp.

Green, T. R. G., and Petre, M. 1992. When visual programs are harder to read than textual programs. *ECCE-6: Proceedings of the Sixth European Conference on Cognitive Ergonomics*, September 1992.

Ichikawa, T., and Hirakawa, M. 1990. Iconic programming: where to go? *IEEE Software* 7(7): 63–68.

Kiper, J. D., Howard, E., and Ames, C. 1996. Criteria for evaluation of visual programming languages. *Journal of Visual Languages and Computing* 8(2) 175–192.

Moher, T., Mak, D. C., Blumenthal, B., and Leventhal, L. M. 1993. Comparing the comprehensibility of textual and graphical programs: the case of Petri nets. *Empirical Studies of Programmers: Fifth Workshop*, C. R. Cook, J. C. Schotz, and J. C. Spohner (eds.), 137–161.

Myers, B. A. 1990. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing* 1(1): 97–123.

Pandey, R., and Burnett, M. 1993. Is it easier to write matrix multiplication visually or textually? An empirical study. *Proceeding of 1993 IEEE Symposium on Visual Languages*, 344–351.

Petre, M., and Green, T. R. G. 1993. Learning to read graphics: some evidence that 'seeing' an information display is an acquired skill. *Journal of Visual Languages and Computing* 4: 55–70.

Petre, M. 1995. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM* 38(6): 45–56.

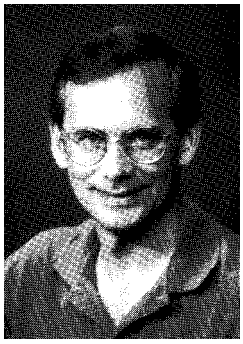
Price, B. A., Baecker, R. M., and Small, I. S. 1993. A principled taxonomy of software visualization. *Journal of Visual Languages* 4(3): 211–266.

Scanlan, D. A. 1989. Structured flowcharts outperform pseudocode: an experimental comparison. *IEEE Software* 6(5): 28–36.

Shneiderman, B., Mayer, R., McKay, D., and Heller, P. 1977. Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM* 20(6): 373–381.

Von Mayrhauser, A., and Vans, A. M. 1995. Program understanding: models and experiments. *Advances in Computers* Vol. 40. M. C. Yovits and M. V. Zelkowitz (eds.). San Diego: Academic Press, 1–38.

Vose, G. M., and Williams, G. 1986. LabVIEW: Laboratory Virtual Instrument Engineering Workbench. *Byte* September: 84–92.



James D. Kiper is an associate professor in the Department of Systems Analysis at Miami University, Oxford, Ohio. He received the M.S. and Ph.D. degrees in Computer Science from the Ohio State University. His research interests are software engineering, visual programming languages, and human-computer interaction.

Brent Auernheimer is a professor of computer science at California State University, Fresno. He received B.A., M.S., and Ph.D. degrees from the University of California, Santa Barbara. He received ASEE and AWU fellowships for research at NASA's Kennedy Space Center and the Jet Propulsion Laboratory. His research interests are software engineering and human-computer interaction.

Chuck Ames received the Master of Systems Analysis degree from Miami University in 1991, and is now employed as Operations Manager and System Engineer of the Flight System Testbed at the Jet Propulsion Laboratory, California Institute of Technology. Chuck's research interests include visual programming, distributed real-time systems, and web-based collaborative information systems.