Visual Programming Languages and the Empirical Evidence For and Against

K. N. Whitley

Department of Computer Science
Vanderbilt University
Box 1679, Station B
Nashville, TN 37235

whitley@vuse.vanderbilt.edu

October 1996

To appear in *Journal of Visual Languages and Computing*

Contents

Abstract

The past decade has witnessed the emergence of an active visual programming research community. Yet, there has also been a noteworthy shortage of empirical evidence supporting the resulting research. This paper summarizes empirical data relevant to visual programming languages, both to show the current empirical status and to act as a call to arms for further empirical work.

1 Introduction

The past decade has witnessed the emergence of an active visual programming research community whose efforts have yielded many visual programming languages (VPLs) and visualization systems. To date, there has also been a shortage of empirical studies backing the design decisions in these VPLs and visualization systems. Definite negative consequences stem from this lack of evidence. For example, critics and the larger computer science community are more inclined to dismiss visual programming as an unpromising research fad. In addition, visual programming researchers are forced to make design choices without good data to direct their decisions. Given the unproven status of visual programming, empirical studies could make a major contribution to visual programming efforts. This is so much the case that the empirical evidence problem warrants being called the biggest open problem in visual programming research. At the least, the evidence problem deserves equal footing with the scalability problem, which has been recognized by many visual programming researchers as a major open problem [1].

Fundamentally, visual programming research rests on the idea that visual notations can be properly integrated with text to form VPLs and visualization systems that will improve the human programming experience. This paper asks, "What data exists that tells us when and how visual notations can be beneficially used within the context of the complex cognitive activities required by programming?" This paper focuses on VPLs and, to a lesser extent, on visualization systems. The visual notations of interest are depictions (such as lines, patterns and geometric shapes) that form the non-textual parts of a VPL or that closely resemble non-textual parts of a VPL. For brevity's sake, the term *visuals* is used through this paper in place of the term visual notations.

The remainder of this paper examines this empirical data as follows. Section 2 starts the investigation with discussion of some issues that are raised when one considers performing proper user studies. Section 3 presents evidence that reflects favorably on a particular visual or VPL,

while Section 4 summarizes work that found no benefits accruing from the studied visual. These two evidence sections include studies conducted by the visual programming community, as well as pertinent studies from the broader computer science literature and from cognitive psychology. In all cases, this paper notes the studies' contributions and, in some cases, design flaws. Interested readers can also consult the unabridged version of this paper [2]. Finally, Section 5 concludes with a look to the future.

2 Issues

2.1 Need for Empirical Work

In one sense, the scarcity of empirical studies of visual programming is not surprising. Software engineering and programming languages are areas that historically have suffered from a lack of empirical validation. A common pattern in software engineering research is the development of system-building techniques, such as object-oriented design, which are strongly advocated in the absence of evidence. Part of this situation stems from the difficulties of carrying out studies on projects the size of typical software projects. However, another factor is the resistance of computer researchers to adopt techniques outside of the traditional set of computer science techniques. Recently, strong arguments have appeared that acknowledge the difficulty of performing quantitative assessments, yet that urge the software community to overcome its inertia [3, 4, 5]. The bottom line is that software engineering is an empirical field. Ultimately, any software engineering technique or theory of programming is judged by whether it produces cost-effective results.

Designing a programming language involves a mixture of factors, which ideally includes an understanding of the cognitive demands required by the language. Clearly, visual programming fits into this category of topics to which empirical techniques apply. Like software engineering in general, visual programming suffers from a need for empirical data [6, 7, 8, 9, 10]. By surveying the visual programming literature, the lack of empirical foundation becomes apparent; the majority of the research is backed by unvalidated claims. A recent paper by Blackwell catalogs the hypotheses expressed throughout the visual programming literature [11]. Even though some claims are reasonable initial hypotheses, the shortage of supporting evidence leaves the door wide open to justifiable attacks from critics. So, Green dismisses some claims as "ill-digested pseudo-psychology" [9], and

Brooks predicts that the entire field will ultimately prove useless [12].

To promote a balanced view, it is worth mention that the software engineering and programming communities are not completely without an empirical tradition. Some of the studies summarized below are proof of this fact. Moreover, there are workshops directed at empirical study of programming, including the Empirical Studies of Programmers (ESP), the IEEE Workshops on Program Comprehension and the Psychology of Programming Interest Group (PPIG) annual workshop. However, use of empirical techniques is currently more the exception than the rule.

2.2 Methodological Difficulties

Researchers, whether interested in undertaking empirical studies or simply in assessing the studies of others, need an understanding of methodological issues. In fact, the software engineering community as a whole needs to boost its understanding so that future studies are not hampered with the flaws that have plagued previous studies. Fenton, Pfleeger and Glass [3], Vessey and Weber [13], Sheil [14], Brooks [15] and the *Psychology of Programming* [16] all provide excellent explanations of some of the biggest methodological problems. These explanations are not repeated here, with the following few exceptions.

In conducting empirical user studies, there are two major areas of concern: theory and experimental design. While the theory and design phases overlap in issues, there are distinct issues raised in each. Understanding the difference between the two is a key to understanding the methodology of empirical studies. Vessey and Weber [13] succinctly explain the crux of the theory issues: "Good theory is a prerequisite to good empirical work...." A researcher designs a given empirical study to test a hypothesis. Ideally, that hypothesis is generated from an underlying theory. On the strictly atheoretical extreme, an empirical study can be performed without a theoretical underpinning, but the utility of such a study is limited. In contrast, a theory provides guidance in targeting what behavior to study and in assessing a study's results. Vessey and Weber [13] discuss this topic in terms of hypotheses for prediction versus hypotheses for understanding; Moran [17] discusses this topic by describing four points along an empirical-theoretical spectrum. A current problem for programming studies is the rudimentary nature of relevant cognitive theory. Programming boils down to a complex form of problem solving, involving ill-structured design problems. In fact, there is relatively little research on ill-structured problem solving in general.

After experimental hypotheses for a study have been formed, experimental design issues come to the fore. Care is required in all aspects of the experimental design including the selection of subjects, choice of experimental task, identification of the dependent variable(s) and choice of meaningful statistical techniques. One particular issue in studies of programming (as with studies of other complex problem-solving tasks) is the problem of large individual differences. Previous studies of programmers have shown that programmers are a very heterogeneous population, a fact that can create difficulties when the effects of an independent variable(s) are masked by the differences within the population. For example, in a between-subjects experimental design, subjects are divided into groups, and the different groups are exposed to different experimental conditions. If the variability of scores within each group is large relative to the variability between the groups, a real effect can be hidden. There are two implications of this problem. First, researchers often need to take special measures in the design of their experiments. One technique that has been successful in several studies is the use of within-subjects experimental designs; by testing each subject with all of the experimental stimuli under study, the statistical analysis is based on the relative performance of each condition for each subject. Second and possibly more importantly, the large individual differences are largely unexplained and, thus, raise research questions of their own; an understanding of what distinguishes individual programmers will be an integral part of a well-developed theory of programming. Moreover, given the magnitude of the individual differences, an understanding of the source(s) of these differences may yield real gains in programming productivity.

2.3 Visual Programming: Possible Effects

In setting an agenda for empirical assessment of VPLs, one should develop an initial theory of how VPLs might affect programming. A perusal of the initial studies of VPLs reveals that several of the studies test relatively vague hypotheses; one common hypothesis is that notations that properly incorporate visuals allow people to complete tasks faster than purely textual notations. These hypotheses do not go beyond this general speculation to more detailed suppositions of why the visual would cause a speedup or how this would affect the software lifecycle. This subsection presents a preliminary analysis intended to provide some structure to a discussion of the reasons behind possible effects of VPLs.

An initial question might be whether notation plays a significant role in human performance.

Researchers do know that problem-solving performance is affected by the way in which a problem is presented. As several of the studies described below show, people frequently do not translate a given problem into a normalized internal (mental) representation that is independent of the original external presentation. Rather, external representations tend to constrain internal representations. A corollary is that people do not always naturally or instinctively use efficient representations. Thus, there is reason to speculate that visual notations affect programming at some level. However, to develop theory and design an experiment, one must push toward uncovering specifics of the phenomena. One strategy used several times by the researchers in the studies described below is to test visual notations relative to familiar phases in the software lifecycle. Thus, they tested notations incorporating visuals in the context of specific tasks, such as design, coding, debugging and modification tasks. This partitioning of programming tasks comes easily, as it follows an accepted understanding of programming.

An alternative framework comes by considering cognitive processes. Within each programming task, at least two cognitive skills are required: searching and comprehension. Using the analogy of a production system, Larkin and Simon break the problem into three cognitive skills: search, recognition and inference [18]. Thus, the question arises as to whether VPLs actually affect people's ability to understand and solve a problem or whether VPLs solely affect a more routine skill such as searching. An example of the latter case would be if VPLs provide perceptual cues that allow a person, when searching for a piece of information, to skip over portions of code that would otherwise have to be examined. There might be benefits in either case, although there may be implications about which audience could use VPLs effectively. Some researchers have hypothesized that VPLs will be successful at empowering end-user programming. For VPLs to succeed in end-user programming, one can reasonably surmise that VPLs must ease the problem-solving burden. In contrast, if VPLs primarily speed search, then the proper audience might be trained programmers. In this case, VPLs may be practical, as searching may represent a substantial portion of programming time for certain types of programming.

One body of work certain to factor into future developments in the theory of VPLs is the cognitive dimensions framework put forward by Green and Petre [19, 20]. Green and Petre have set forth a set of vocabulary terms, known as cognitive dimensions, designed to describe the cognitive

design space of information structures (such as programming languages). Their astute observations provide a basis for a theory. Moreover, the cognitive dimensions can be used early in the design process to evaluate VPL designs. For examples of how the cognitive dimensions apply to visual programming, consult [19] for evaluations of VPLs including LabVIEW and Prograph, [21] for an evaluation of the VPL Pursuit and [22] for an evaluation of spreadsheets.

Finally, when developing an initial theory of the effects of VPLs, there are two cautions for researchers to consider. First, previous studies of programming notations have not established language features as large factors in programming performance. In particular, textual syntactic issues do not seem to be a bottleneck in software development for trained programmers using high-level languages. Thus, to establish a theory of the effects of VPLs, it might pay to examine whether/how VPLs address more than syntactic issues. Second, in cases where significant effects due to notation have been found, the magnitude of the effects may not warrant much enthusiasm. Experimental studies can be designed to detect minute differences between the experimental conditions. Thus, significant results do not necessarily mean that the effect size is large enough to yield cost-effective performance gains.

3 Evidence For

An underlying premise of visual programming research is that visuals will improve the human-machine interface by catering to human cognitive abilities. Currently, only a small body of empirical evidence exists to back this premise. All of the studies discussed in this section are part of this body of evidence. As an overview, Table 1 lists these studies in the order in which they are discussed.

3.1 Non-Programming Studies

3.1.1 People perform better with organized and explicit information

An accurate generalization explaining the relative efficacy of alternative representations is that people perform better when presented with information arranged in a consistent and organized manner. Furthermore, the more efficient representations tend to be the ones that make information explicit. These two principles apply to all forms of representation, textual as well as visual. For example, when given a textual list of information, people usually process the information more effectively if the list is alphabetized or semantically grouped. With regard to visual representations, there

Researcher(s)	Reference(s)	Section	Representation(s)
Day	[23]	3.1.1	list vs. matrix, list vs. spatial map
Schwartz, Fattaleh, Polich	[24, 25, 26]	3.1.1	sentence vs. tree vs. matrix
Carroll, Thomas, Malhotra	[27]	3.1.1	matrix
McGuinness	[28]	3.1.2	tree vs. matrix
Scanlan	[29]	3.2.1	pseudocode vs. flowchart
Vessey, Weber	[10]	3.2.1	pseudocode vs. tree vs. decision table
Cunniff, Taylor	[30]	3.2.1	Pascal vs. FPL
Pandey, Burnett	[7]	3.2.2	Pascal vs. Modified APL vs. Forms/3
Baroth, Hartsough	[31]	3.2.3	C vs. LabVIEW, VEE

Table 1: Studies with results favorable to a visual notation. These studies are discussed in Section 3; they are listed here in the order in which they are discussed. Each entry in the table represents a study or group of related studies. In this table, the Researcher(s) column lists the researcher who conducted a study. The Reference(s) column lists the full study report. The Section column gives the section of this paper in which the study is discussed. The Representation(s) column lists the textual and visual notations under investigation.

is evidence to support the hypothesis that, sometimes, the organization made possible by visual notations allows more effective displays than are possible with textual notations. For example, Day has studied the effects of alternative representations in many different domains, including the domains of bus schedules, medication instructions and computer text editing [23].

In the case of medication instructions, Day devised two representations of a set of instructions for a stroke patient. The instructions direct the patient to take 15 pills (distributed over 6 different drugs) over four times of day. Then, Day exposed different subject groups to the different representations. After studying the instructions, subjects answered questions about which type and how many pills to take at which times of day. The study was a 2×2, between-subjects design. A 2×2 study is an experiment with two independent variables, each having two values, in which all four combinations of the two independent variables are tested. In a 2×2, between-subjects design, four groups are measured along the dependent variable. Day's first independent variable was representation. Two groups of subjects were given medicine instructions in list (textual) format, while the other two groups dealt with a matrix (visual) format (Figure 1). The second independent variable was memory versus comprehension. Two groups did not have the instructions available during the test phase (the memory condition), while the two other groups did have the instructions to consult (the comprehension condition). The dependent variable was correctness.

List Notation

Inderal	—	1 tablet 3 times a day
Lanoxin	—	1 tablet every a.m.
Carafate	—	1 tablet before meals and at bedtime
Zantac	—	1 tablet every 12 hours (twice a day)
Quinaglute	—	1 tablet 4 times a day
Coumadin	—	1 tablet a day

Matrix Notation

	Breakfast	Lunch	Dinner	Bedtime
Lanoxin	✓			
Inderal	✓	✓	✓	
Quinaglute	✓	✓	✓	✓
Carafate	✓	✓	✓	✓
Zantac		✓		✓
Coumadin				✓

Figure 1: Day's representations for medication schedules. The list format is an actual set of instructions given by a doctor to a stroke patient. (Adapted from [23], page 275, by permission of Academic Press, Inc.).

Significant performance differences resulted for both dimensions. Subjects using the matrix format were correct on 78% of the questions, while subjects using the list were correct on 56% of the questions. Of special note, even subjects in the comprehension group still made errors; this was more so for subjects using the list representation. In discussing the results, Day theorizes that the matrix representation facilitates learning because the matrix allows two dimensions of information to be emphasized simultaneously, in this case heightening visibility of both the medications and times of day. About the list, Day speculates, "It is not clear whether extended use of the list format will produce comparable improvements in performance; if it does, it should still take longer to achieve such performance, with more opportunity for harmful errors during this time."

Day's study of computer text editing is another example of visual outdoing textual. Subjects were observed to see how quickly they could learn a small set of six elementary cursor-movement commands and how efficiently they could apply the commands. A 2×2, between-subjects study was designed, the independent variables being representation and symbol-definition match. The representations were an alphabetized list (textual) and a spatial map (visual); see Figure 2. The symbol-definition match dimension relates to the principle of stimulus-response compatibility; it is not germane to the current discussion. The dependent variables were correctness and efficiency. Day also took into account the subjects' prior experience with editors. The study involved a memorization phase followed by a testing phase. During the memorization phase, subjects studied their assigned representation for two minutes. The testing phase involved a problem-solving task of 10 on-paper problems requiring a subject to move a cursor from its given location to an indicated,

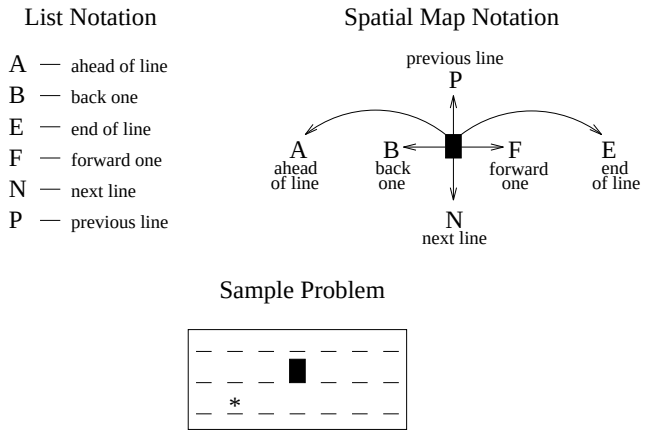

Figure 2: Day's representations for text editing commands. The sample problem shows what the subjects saw during the testing phase: a simplified computer screen in which the dashes indicate possible cursor locations, the box represents the current cursor position, and the asterisk represents the goal position. (Adapted from [23], page 286, by permission of Academic Press, Inc.).

new position. Time allotted to complete the problem solving was fixed to two minutes.

As for the correctness of the problem-solving section, subjects using the spatial map had a significant advantage. They scored 91% correct versus the 77% correct scored by the groups using the list format. Prior knowledge of editors had no effect on the correctness scores. Even more interesting than the correctness results is the analysis of problem-solving efficiency. When subjects solved particular problems correctly, they did not always do so efficiently. Day measured efficiency in terms of excess keystrokes. For the three problems in which excess keystrokes were possible, subjects using the list format averaged 2.1 excess keystrokes on correctly-solved problems, while those using the spatial map averaged only 1.2 excess keystrokes. In contrast to the correctness results, prior knowledge of editors affected efficiency. Subjects with low prior knowledge averaged 2.3 excess keystrokes, while those with high knowledge averaged 1.7. Day remarks on subsequent work of hers that expands on the initial study; the same command set was used, but the problem-solving was more difficult. The initial results were magnified in the expanded study: the number of excess keystrokes given by subjects using a list format was over 12 times more than those using a spatial map. In sum, Day's work in text editing supports two hypotheses. First, visual representation can make a difference even in small-sized problems. Second, the benefit of using visual representation can grow as the difficulty of the problem grows.

Day's studies give weight to the principle that visuals can facilitate the presentation of informa-

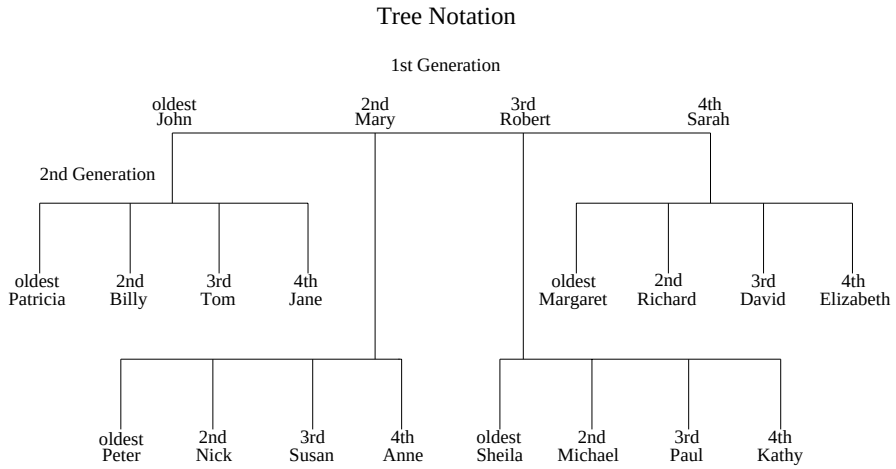

Figure 3: McGuinness' tree representation for family relationships. The actual version given to the subjects had the second generation laid out in one level across the page. (Adapted from [28], page 271, by permission of Psychonomic Society, Inc.).

tion in a consistent, explicit and organized fashion. Similarly, Schwartz and Fattaleh [26, 25] report that a matrix outperformed sentence rewrites and tree diagrams for solving deductive reasoning problems. Polich and Schwartz [24] add that the superiority of the matrix grew as the problem size grew. Carroll, Thomas and Malhotra [27] conducted a performance study in which subjects were asked to solve relatively complex design problems. Subjects solved one of two isomorphic problems. In one experiment, subjects performed better when solving the spatial isomorph (an office layout problem) as opposed to the temporal isomorph (a scheduling problem). However, in a second experiment, the performance differences were mitigated when subjects were trained to use a matrix representation.

3.1.2 Notations are not superior in an absolute sense

Delving deeper into the questions of when and why visuals affect problem-solving, McGuinness conducted a study comparing two visual representations [28]. McGuinness' study involves the principle that notations are probably not superior in an absolute sense; rather, they are good in relation to specific tasks[1]. McGuinness used tree notation and matrix notation to encode information about family relationships (Figures 3 and 4). The relationships, which included parent-child and cousin relationships, were for a set of 20 family members. Based on this family information, two sets

[1]This principle is known as the match-mismatch hypothesis and will also be discussed in Section 4.

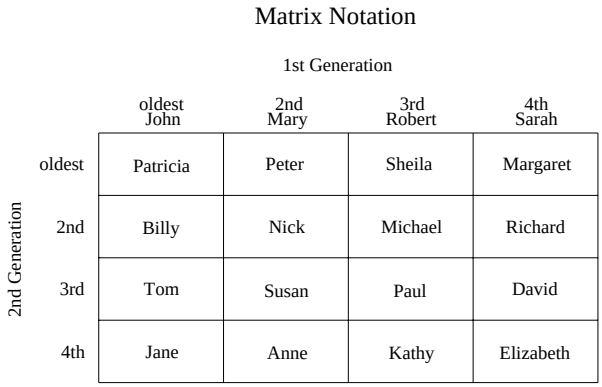

Figure 4: McGuinness' matrix representation for family relationships. (Adapted from [28], page 272, by permission of Psychonomic Society, Inc.).

of questions were devised. One set focused on the cousin relationships and posed questions about who could go on vacation with whom. The other set concentrated on sibling relationships and asked questions about who could inherit whose house. Correct answers were restricted by rules and exceptions. A rule specified the sex and the birth ordering (older/younger) of a valid vacation companion (or inheritor). An exceptions list named one to four people who were not willing to go on vacation (or who had been disinherited). For both question sets, problem-solving was largely a matter of searching the given knowledge base.

A mixed-factors design was used, which involves both between-subjects and within-subjects variables. Independent variables relevant to the current discussion were representation (a between-subjects variable) and number of exclusions (a within-subjects variable). Subjects were assigned to one of two groups (the tree group or the matrix group). Both groups memorized their representation before the testing phase and answered both sets of questions. The dependent variable was the time needed to answer a question.

To examine the results, consider performance on the two question sets separately. For the vacation questions, the effects of both independent variables were significant. The representation had a dramatic impact. Subjects who learned the tree notation took more than twice as long as subjects who learned the matrix. There was no speed/accuracy tradeoff. Furthermore, as one would intuit, the number of exclusions significantly affected performance. In fact, there was significant interaction between the number of exclusions and the representation; questions with more exceptions became increasingly more difficult for the subjects using tree notation. Whereas the performance

differential between the notations started at approximately 10 seconds for 1 exception, it had risen to over 50 seconds for 4 exceptions. McGuinness summarizes, "It seems that as questions became more difficult, efficient representation became more important." In contrast, for the inheritance questions, representation did not significantly affect performance. The number of exclusions did affect performance but did so almost identically for both representations. McGuinness interprets this contrast as reflecting differences in how the family ties were mapped onto the two notations. Both the tree and the matrix contained the same information, yet the matrix allowed one to stress two dimensions simultaneously. The matrix and the tree provided fairly direct representations of the sibling relationships, but the matrix also simultaneously allowed direct access to the sort of cousin information required to answer the vacation questions.

To probe deeper, McGuinness queried subjects about how they had answered the questions. The subjects described a process of mentally stepping through the notation in a systematic fashion and crossing out ineligible elements. From these descriptions, McGuinness proposed a way to quantify the mental effort required by the questions. McGuinness defined the concept of a mental step as one access to a notation element. McGuinness totalled the mental steps required by each question in each of the two notations. Then, for evidence to support the mental-step hypothesis, McGuinness conducted a second study. If the mental step analysis is valid, then certain changes to the information mappings should result in predictable changes in response times. McGuinness performed such a study by altering the way in which the same family information was encoded in the notations; the cousin relationship was stressed in both altered notations, whereas sibling relationships were stressed in only the matrix. Given the same two sets of questions, the step counts predicted that the inheritance questions would become more difficult for the tree notation and that the vacation questions would not yield significant differences. The results of the study did match the predictions, thus strengthening the idea that notations and mappings are often efficient only in relation to specific processing demands.

3.2 Programming Studies

3.2.1 Flowcharts as a notation for control flow

Historically, the main visual tool available to programmers has been flowcharts. Thus, previous studies often focused on whether flowcharts increase comprehension[2] compared to standard program text. A recent example is Scanlan's work [29]; Scanlan devised an experiment to detect differences in the comprehension of conditional logic expressed in structured pseudocode versus structured flowcharts (Figure 5). The testing procedure entailed subjects' viewing conditional logic and then answering questions about the boolean states required to trigger certain actions. The independent variables were representation and algorithm complexity. The study used three algorithms, one each of simple, medium and difficult complexity. Each of the three algorithms had both a flowchart and pseudocode variation, for a total of six stimuli. The study used a within-subject design, meaning that each subject encountered all independent variable values and performed all tasks. The apparatus for the experiment was designed so that sensitive timing data could be isolated. Specifically, there were five dependent variables: the time taken to comprehend an algorithm, the time spent answering questions about an algorithm, the number of times a subject brought an algorithm into view, the answers to all questions (i.e., correctness) and the subjects' confidence in their answers. Subjects were given as much time as they needed to answer the questions. The subjects were MIS (management of information science) students.

Save for one exception, structured flowcharts had a significant advantage for all dependent variables and for all the levels of algorithm complexity. The most sensitive measure was the time needed to comprehend an algorithm. Scanlan was surprised to find that flowcharts made a difference even in the simple case. Furthermore, the benefit of the flowcharts increased as the algorithms became more complex. The ratios of the time needed to comprehend the pseudocode compared to flowcharts were 1.7, 1.9 and 2.5 for the simple, medium and complex cases. These results are dramatic considering that the students using pseudocode also produced significantly more errors.

In assessing Scanlan's results, observe that Scanlan's study was well-designed to isolate possible effects of the independent variables. Scanlan carefully isolated timing data, and the results

[2]In many studies, including all of those summarized here, the term comprehension is used to refer to subjects' performance in answering questions (i.e., about a segment of programming code) as opposed to creating solutions (i.e., writing a segment of code); in these cases, it is not used to distinguish between specific cognitive skills (e.g., comprehension versus searching) as was done in Section 2.3.

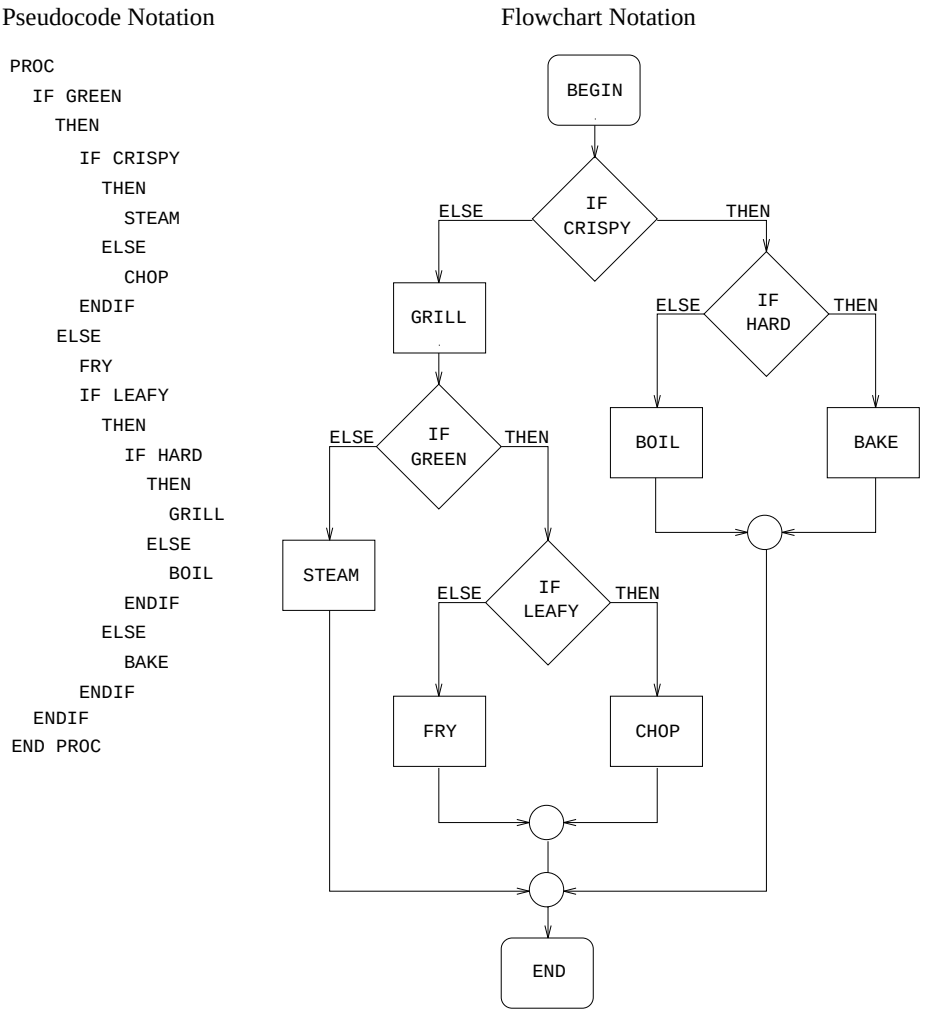

Figure 5: Scanlan's representations for conditional logic. This pair gives two different examples, both at the medium level of algorithm complexity. (Adapted by permission of the publisher, from [29], page 30, ©1989 IEEE).

indicate that some prior studies of flowcharts were flawed to the extent that effects of representations may have been masked (this discussion will be raised again in Section 4.2.1). On the other hand, Scanlan's stimuli were simple in the sense that the actions in the branches of the conditional statements were only one-word action verbs (like "grill" and "bake"). The conditions were also one-word adjectives. So, whereas Scanlan's structured flowcharts excelled in illuminating conditional logic, his flowcharts did not have detailed actions typical in real programs (involving boolean expressions, arithmetic expressions, and variable dependencies). Nor did the programs involve iteration. Furthermore, the questions asked pertained to control flow, which is the type of information that flowcharts highlight. Finally, the particular pseudocode layout may have biased the study

against the pseudocode; namely, the programming style of spreading one if-then-else statement over six lines may have made the pseudocode examples harder to read compared to more compact professional programming styles.

Scanlan's study does suggest, however, that flowcharts can have a beneficial effect for certain tasks. Similar results occurred in studies by Vessey and Weber [10] and by Cunniff and Taylor [30]. Vessey and Weber compared decision trees, pseudocode and decision tables in the task of expressing conditional logic. In terms of the time needed to perform the experimental tasks, decision trees performed equal to or better than pseudocode, while both outperformed decision tables. Cunniff and Taylor compared Pascal to FPL, a flowchart-based VPL aimed at beginning programmers, in the task of comprehending small code segments. Beginning programmers took significantly less time answering questions about FPL code segments. However, in their reporting of the study, Cunniff and Taylor do not provide example questions, nor is it clear what the complexity of the code segments was. As Green, Petre and Bellamy note, "Regrettably, [Cunniff and Taylor] did not report their experimental procedure in enough detail for the reader to be fully certain what they have shown!" [32, page 129].

3.2.2 Forms/3 as a notation for matrix manipulations

Currently, a study conducted by Pandey and Burnett stands out as the strongest controlled study favorable to VPL [7]. Their study is particularly interesting since they compared two textual programming languages to a non-flowchart VPL. Specifically, Pandey and Burnett tested subjects' performance on vector and matrix problems using three representations: Pascal, a modified form of APL[3] and the VPL Forms/3 (Figure 6). The experiment used a within-subject design. The independent variable was representation (the three languages), and the dependent variable was correctness. The subjects were CS students; of these, all had experience with Pascal and/or C; one had experience with APL; none had experience with Forms/3.

The experiment consisted of a lecture followed by a testing phase. In the lecture, subjects went through six vector/matrix problems and were shown answers to each in all three languages. All the concepts needed for the vector/matrix manipulations in each language were presented.

[3]They made two APL modifications: They used an alternate syntax that is English-based, as opposed to APL's use of special mathematical symbols, and they used left-to-right evaluation as the rule for expression evaluation.

Pascal

```
procedure fib
      (var v: Vector; n: integer);
var i: integer;
begin
     v[1] := 1;
     v[2] := 1;
     for i := 3 to n do
        v[i] := v[i-1] + v[i-2];
end;
```

Modified APL

```
v = 1 1
temp1 = Take(2, R, v)
temp2 = RowReduce(+, temp1)
v = Append(C, v, temp2)
if Dimension(C, v) <> N then 2
```

Forms/3

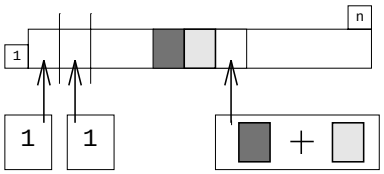

Figure 6: Pandey and Burnett's representations for vector and matrix manipulation tasks. This trio shows the Pascal, modified APL and Forms/3 solutions to the problem of computing the first N elements of the Fibonacci sequence. (Adapted by permission of the publisher, from [7], page 350, ©1993 IEEE).

Furthermore, the subjects were given copies of the lecture overheads before the lecture began; they could take additional notes and could refer to the copies/notes during the test. Thus, whereas other studies used memorization as a way of measuring the effect of a representation, Pandey and Burnett wanted the note-taking to improve learning, and they wanted to remove memorization as a factor on performance. In the testing phase, subjects created solutions to two matrix problems using all three languages, for a total of six solutions. One problem was to append two compatibly-shaped matrices together. The other was to compute a sequence of Fibonacci numbers. The solutions were done on paper. Subjects were allotted five minutes to work on each solution.

Correctness was assessed by assigning each solution a rank based on a four-point ordinal scale. Correctness was based on semantics; syntax errors were ignored. The results are surprising. Forms/3 stood out for both problems. On the append problem, Forms/3 and APL outperformed Pascal. Of particular relevance is the fact that APL had a built-in append operator which had been presented in the lecture. On the Fibonacci problem, Forms/3 and Pascal outperformed APL. Over both problems, 73% of the Forms/3 solutions were completely correct compared to 53% of the APL solutions and 40% of the Pascal solutions. The results are surprising because they go against intuition in two ways. First, the problems were short and simple, leading the authors to initially fear

that ceiling effects would occur (i.e., that any experimental differences would be masked because the task was too simple). Second, given the subjects' prior experience with Pascal and/or C, one would suspect a bias towards Pascal. Yet, even though all but six of the subjects had previously seen or written a Fibonacci solution in Pascal and/or C, Forms/3 still performed comparably.

3.2.3 LabVIEW and its use in participatory programming

The last study in this section is an industry-based, observational study from the Measurement Technology Center (MTC), a group that supports the work of the Jet Propulsion Laboratory [31]. The authors, Baroth and Hartsough, claim productive use of VPLs in real programming situations. Their assessment stems from results of a study using LabVIEW and from further observations culled from several years of MTC's use of LabVIEW and VEE for building test and measurement systems. LabVIEW and VEE are both commercial VPLs based on the dataflow paradigm. Both are also targeted for end-user programming, namely to enable scientists and engineers to write data acquisition, analysis, display and control applications.

The study involved parallel development of a telemetry analyzer by two development teams, the goal being to discern how LabVIEW would compare to conventional, text-based programming. A LabVIEW development team was compared with a simultaneous effort of a text-based (C) team. Both development teams received the same requirements, the same funding and the same amount of time (three months) to complete the project. At the end of the three months, the LabVIEW team was much more advanced than the C team. Whereas the C team did not complete the requirements, the LabVIEW team had gone beyond the original requirements. The LabVIEW performance convinced people within MTC of the viability of visual programming. Consequently, the LabVIEW group was awarded additional projects.

Baroth and Hartsough attribute the productivity benefits of VPLs to increased communication between the customer and developer, which they in turn attribute to the visual syntax of the VPLs. In particular, their customers are engineers and scientists who possess limited programming experience, yet who are comfortable with dataflow diagrams. Thus, LabVIEW and VEE programs are visually familiar to the MTC customers. Baroth and Hartsough consistently find that their customers understand much of the process in LabVIEW or VEE code (albeit, not necessarily the details of the code). Baroth and Hartsough explain that this accessibility of the visual syntax allows

the customer to become more involved in the development process. Instead of communicating solely via formal written requirements documents and in meetings, the customer and developer can jointly develop parts of a system. Baroth and Hartsough report that their customers understand the dataflow diagrams well enough to make suggestions and to point out errors. Thus, the customer can sit by the developer during system building, especially during the initial phases when the inevitable questions about the exact nature of the requirements arise.

In fact, this participatory-prototyping style was used by the LabVIEW team and can be seen as a confounding factor in that study. During the study, the LabVIEW team met frequently with and interactively coded with their customer. The C team followed a classical approach; they worked from the initial requirements and did not meet with the customer until they demonstrated their system at the end of the three-month time period. Thus, the results of the study are possibly biased in favor of LabVIEW; the C team might have performed better had they used a rapid-prototyping strategy and met frequently with their customer. On the other hand, this study was not intended as a tightly-controlled experiment. So, whereas one cannot conclude anything specific about the visual aspects of LabVIEW, this study is helpful in two ways. First, this study is evidence that industry can make beneficial use of commercial VPLs in real programming situations. Second, Baroth and Hartsough's observations that their customers easily understood much of LabVIEW and VEE code yields a direction for further research.

3.3 Summary of the Evidence For

People in design and problem-solving situations perform better when information is presented in a consistent and organized manner. Furthermore, the more efficient representations tend to be the ones that make information explicit. These two guidelines apply to all notations, including textual ones. The studies in this section indicate that, compared to textual notations, visual notations can provide better organization and can make information explicit. Moreover, properly-used visuals result in quantifiable performance benefits. Several studies showed visuals outperforming text in either time or correctness, sometimes in both.

Notations are probably not good in an absolute sense. As Green writes, "A notation is never absolutely good, therefore, but only in relation to certain tasks" [20]. This situation is observable within McGuinness' study of a tree versus a matrix notation. It can also be observed across studies.

The matrices used in McGuinness' study outperformed trees in certain situations, yet the trees in Vessey and Weber's study consistently outperformed the decision tables (i.e., matrices). In this particular comparison across these two studies, one possibility that explains when a matrix was helpful is the characteristic of systematic access. In the McGuinness study, the matrices were faster to use than the trees when the question set allowed the subjects to search the matrix in a consistent row-by-row or column-by-column fashion that progressed from one element to a physically-adjacent element. This systematic access pattern was missing from the Vessey and Weber decision tables; also, in parts of the McGuinness study, this systematic pattern of adjacent elements was equaled in the trees. In these cases, the matrices failed to outperform the trees.

It is noteworthy that the summarized studies largely dealt with visuals representing "nonspatial" concepts (e.g., the cooking instructions in Scanlan's study). These studies counter the speculation that visuals are beneficial only when representing objects that have concrete counterparts in the real world. This idea might arise from the observation that highly successful uses of visuals often deal with concrete, spatial objects. One such example is the successful use of diagrams to help students solve physics problems ([18] shows Larkin and Simon's rope and pulley example). Another example is Visual Basic; here, the visuals are used to build GUIs which essentially are spatial objects. This idea might also stem from discussions such as [33] in which Raymond speculates that general programming deals with discrete, not analog, information and thus is not amenable to useful visualization.

On the one hand, the benefit of using visual representations grows as the size or complexity of the problem grows. This trend was observed in four of the summarized studies: Day's editing study, Polich and Schwartz's study, McGuinness' study and Scanlan's flowchart study. So, VPLs may possibly play an important role in traditional programming, where problems are usually larger than the problems used in controlled experiments. On the other hand, visuals can yield better performance even in small-sized problems. This effect was noted in four of the summarized studies: Day's editing study, Scanlan's flowchart study, Cunniff and Taylor's flowchart study and Pandey and Burnett's Forms/3 study. Thus, VPLs may play an important role in end-user programming, where problems are usually smaller than the problems encountered by professional programmers.

Baroth and Hartsough describe a participatory programming style in which end users and

Researcher(s)	Reference(s)	Section	Representation(s)
Wright, Reid	[34]	4.1.1	prose vs. short sentence vs. tree vs. matrix
Shneiderman et al.	[35]	4.2.1	Fortran vs. Fortran + flowchart
Ramsey, Atwood, Van Doren	[36]	4.2.1	PDL vs. flowchart
Curtis et al.	[37]	4.2.1	9 documentation formats
Green, Petre, Bellamy	[38, 39, 9, 32]	4.2.2, 4.2.3	pseudocode vs. LabVIEW
Moher et al.	[40]	4.2.3	pseudocode vs. petri net
Lawrence, Badre, Stasko, Lewis	[41, 42]	4.2.4	text vs. text + algorithm animation, transparencies vs. algorithm animation
Gurka, Citrin	[43]	4.2.4	algorithm animation
Hendry, Green	[22, 44]	4.2.5	spreadsheet
Brown, Gould	[45]	4.2.5	spreadsheet

Table 2: Studies with results unfavorable to a visual notation. These studies are discussed in Section 4; they are listed here in the order in which they are discussed. Each entry in the table represents a study or group of related studies. In this table, the Researcher(s) column lists the researcher who conducted a study. The Reference(s) column lists the full study report. The Section column gives the section of this paper in which the study is discussed. The Representation(s) column lists the textual and visual notations under investigation.

programmers work closely together. If Baroth and Hartsough are correct, then one advantage of VPLs lies in their accessibility to certain classes of nonprogrammers. The questions of whether, why and when VPLs are understandable to end users are certainly good directions for research.

4 Evidence Against

Visuals can simply be poorly designed or be inappropriate for a task. Therefore, visuals do not always provide a benefit and can sometimes encumber the programming process. The studies in this section indicate that the question of whether VPLs are effective for realistic programming remains unanswered. Table 2 lists these studies in the order in which they are discussed.

4.1 Non-Programming Studies

4.1.1 Notations are not superior in an absolute sense

Like the McGuinness study, Wright and Reid performed a study which suggests that one representation may elicit different effects under different circumstances [34]. They were interested in the relative power of prose, short sentences, decision trees and matrices in problem-solving situations. To compare these representations, Wright and Reid devised a series of rules dictating the appropri-

ateness of different space vehicles for space travel. The factors influencing a space vehicle decision included time limitations, cost limitations and length of journey. Wright and Reid, then, made four equivalent versions of the rule base using the four representations.

Wright and Reid tested the four representations using a mixed-factors design. After being presented with the decision algorithm in one of the four representations, each subject was asked a series of 36 problems requiring the subject to choose a mode of space travel appropriate to the problem conditions. The questions were asked in three stages, with 12 questions per stage. In Stage 1, the subjects had the rules available to consult. At the start of Stage 2, the rules were removed without warning. In Stage 3, subjects were told to memorize the rules in preparation for the remaining questions; subjects were given five minutes to learn the rules before the rules were removed. The independent variables of the study were representation (between-subjects), memorization (within-subjects) and question difficulty (within-subjects). The question difficulty refers to whether the question gave the required facts directly or indirectly, the latter type of question considered more complicated. The dependent variables were time and correctness.

Overall, the prose format was consistently more error-prone and slower to use than the non-prose options. However, the more interesting results involved the non-prose options. In Stage 1, there was an interaction between representation (for the non-prose options) and question difficulty. For the easier problems, there were no significant differences in the error rates of the non-prose formats, although the table yielded significantly faster times than any other format. For the harder problems, the decision tree and table gave significantly lower error rates than short sentences or prose. However, the effects changed when subjects had to work from memory. In Stages 2 and 3, there were no overall significant accuracy differences between the representations. Moreover, the distribution of error rates across trials suggests an interesting trend. Over the course of answering questions, the mean percent error for the tree and table deteriorated over trials, while performance with the prose and short sentences improved. A similar distinction between the textual and visual representations occurred in the timing data.

Wright and Reid speculate that the subjective reorganization needed to memorize the textual material gave rise to the improved performance of the prose and short sentences. In any event, along with the McGuinness study, their study indicates that the effect of a notation is relative

to the task that must be performed. Indeed, the question of whether visuals are appropriate for programming is difficult since programming involves many cognitive tasks. The difficulty lies both in ascertaining which tasks account for a significant proportion of programming effort and whether a given VPL benefits those processes.

4.2 Programming Studies

4.2.1 Flowcharts as documentation for control flow

Because flowcharts historically were the primary visual tool available to programmers, more studies have been done on flowcharts than on other visual programming constructs. Two such early studies, one by Shneiderman et al. [35] and the other by Ramsey, Atwood and Van Doren [36], were undertaken at a time when flowcharts were considered to be a notation for documentation (in other words, considered supplemental to textual source code). Neither one found a benefit from using flowcharts as an aid to using textual code.

The Shneiderman et al. study was a series of five experiments testing the effects of flowchart documentation on novice programmers in comprehension, composition, debugging and modification tasks. None of the five yielded significant differences between the flowchart and non-flowchart groups. In assessing this study, however, one must consider several design flaws. For example, [37, 29, 3] all discuss flaws with the Shneiderman et al. work. A primary flaw concerns the way that time was handled. In two experiments, students were given all the time they wanted to complete the work, yet the investigators did not measure time as a dependent variable. In the debugging experiment, time was fixed, but some groups were given more time (no explanation is given). In Scanlan's opinion, "...the time needed to comprehend the algorithm is the most sensitive and important measure [of the effectiveness of a representation]" [29, page 30]. In light of these design flaws, it is not possible to find strong implications in the Shneiderman et al. results. However, the study does raise the question of whether flowcharts are helpful in full-fledged programming tasks.

The Ramsey, Atwood and Van Doren study was a two-phased experiment in which CS graduate students designed and implemented an assembler. The goal of the study was to compare program design languages (PDLs) with flowcharts as notations for the detailed-design phase of software development. The main reported result was significantly better quality and more detailed designs using the PDL. The authors conclude, largely from this result, that PDLs are probably the

preferable design notation. They reason that the subjects tended to include less algorithmic detail in flowchart designs because flowcharts force a person to compress information into a constrained space. Thus, a designer/programmer will probably adopt space-saving habits (e.g., non-mnemonic variable names) that result in lower quality designs. A contrasting opinion, given by Curtis et al. [37], points out that there was a significant skill difference between two groups of subjects that may have confounded the results. Yet, overall, two aspects of this study warrant further attention. First, the control flow information in a program does not seem to be of such overriding import that flowcharts could outperform PDLs. Second, the hypothesized space constraints of flowcharts would be a serious argument against the use of flowcharts.

Currently, the most thorough and definitive work on the value of flowcharts comes in the form of four controlled experiments performed by Curtis and colleagues [37]. Like the other flowchart studies above, Curtis et al. investigated the effectiveness of flowcharts as a documentation tool. Unlike other studies, these researchers undertook a systematic strategy so that they could identify the specific characteristics of a notation that are responsible for the measured effects. So, as opposed to comparing two representations, they first identified two primary dimensions capable of categorizing a wide range of documentation notations. The first dimension, symbology, includes three possibilities: unconstrained text (natural language), constrained text (PDLs) and ideograms (flowchart icons). The symbology dimension measures the succinctness of a notation. The second dimension, spatial arrangement, also has three values: sequential, branching and hierarchical. The spatial arrangements differ in the extent that a notation highlights the execution paths (in this case, the control flow) of a program. Figure 7 illustrates both dimensions. Combining all the possibilities for symbology and spatial arrangement, Curtis and colleagues created nine documentation formats. Figure 8 shows one example of the nine notations. They tested these formats in the context of comprehension, coding, debugging and modification tasks.

One basic experimental design was used across the four experiments. The independent variables were symbology, spatial arrangement and the type of program being documented. Thus, there were a total of 27 experimental conditions. Program type refers to the three different programs used in each experiment: Curtis et al. chose an engineering program (with a complex algorithm), a business program (with a complex data structure) and a program with both a complex algorithm and data

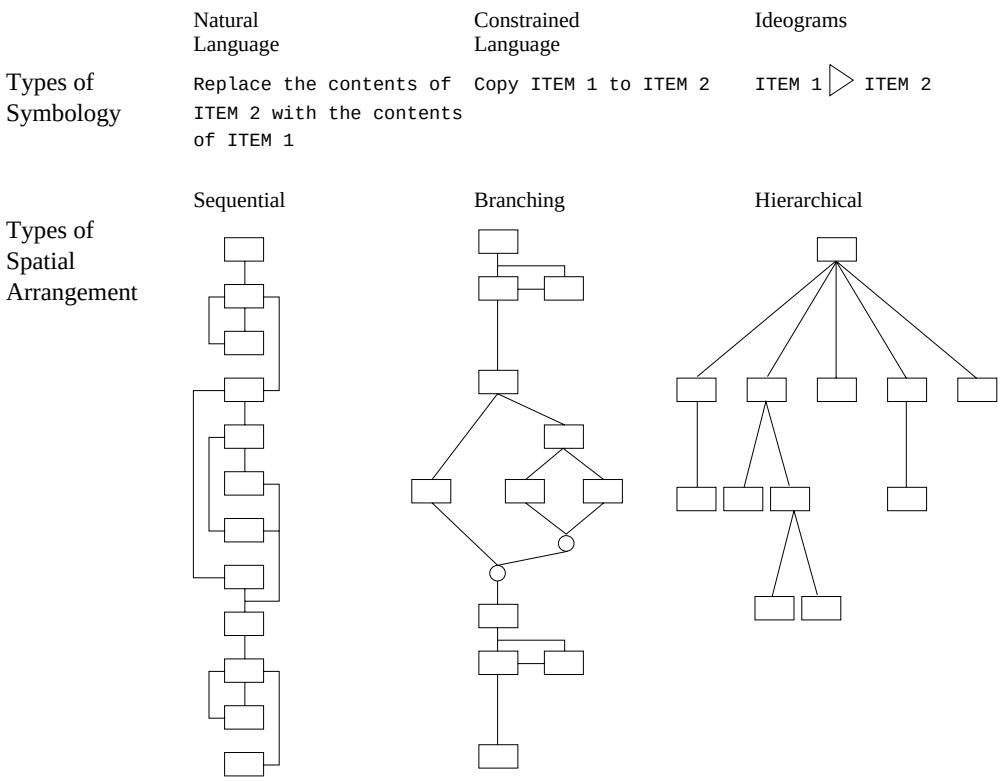

Figure 7: Curtis et al.'s two dimensions that characterize documentation notations. The symbology dimension concerns layout of a program at the level of detail of individual words. The spatial arrangement dimension concerns the layout of a program at the level of program statements (i.e., how statements are juxtaposed). (Adapted by permission of the publisher, from [37], page 173, ©1989 Elsevier Science, Inc.).

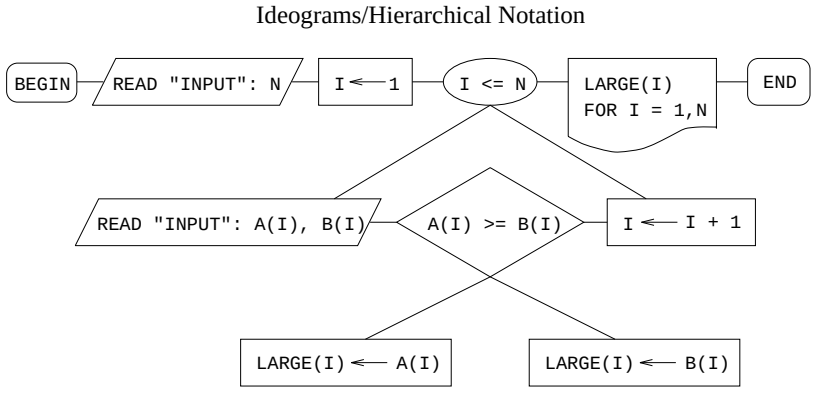

Figure 8: Curtis et al.'s ideograms/hierarchical notation. (Adapted by permission of the publisher, from [37], page 176, ©1989 Elsevier Science, Inc.).

structures. All the programs were coded in roughly 50 lines of Fortran. The dependent variables varied according to the task, although time to complete the task was measured in all cases. In each experiment, each subject performed three tasks, each task involving one of the experimental conditions. All the participants in the experiments were professional Fortran programmers with a minimum of two years of professional experience. Over all four experiments, their average amount of professional experience was over five years.

Across all four experiments, the results were fairly consistent. Furthermore, because the results were so stable across the four experiments, Curtis et al. argue that the results truly reflect the characteristics of the representations studied. For symbology, consistent effects were found in the comprehension, coding and debugging experiments; in the modification study, symbology was not significant, although the data trends were consistent with the other results. In each of these three experiments, symbology was responsible for between 8% to 10% of the variance on at least one dependent variable. These results largely reflect the role of natural language. Prose was the most ineffective symbology format for most tasks, whereas constrained language and ideograms were almost equivalent to one another. For spatial arrangement, only small effects were observed; these occurred in situations in which control flow information was a factor in the task. In particular, significant effects for spatial arrangement were found in the comprehension and modification tasks and accounted for between 2% to 4% of the total variance. These results are largely due to the branching arrangement, which improved performance in some tasks in which control flow was important. Putting symbology and spatial arrangement together, the constrained/sequential representation typically outperformed the other forms of documentation. This format is, in essence, a PDL. The next two runners-up were constrained and ideogram-based symbology presented in a branching arrangement. By far, the largest effect was individual differences, which accounted for between a third and one-half of the performance variations. Curtis et al. consider the magnitude of impact of the individual differences compared to the impacts of the experimental variables to be alarming. Their results are definitely in keeping with the observed trend that individual differences account for a large percentage of performance variation in software productivity.

4.2.2 LabVIEW and the match-mismatch hypothesis

Turning one's focus away from flowcharts, there are few experimental studies of VPLs, much less any solid, replicated results. Green and Petre are two of the few researchers who have undertaken a study of a non-flowchart VPL [9, 32]. In their study, Green and Petre pitted LabVIEW against textual code in a test of comprehension of conditional logic.

Green and Petre's study was not simply a comparison of representation; rather, they made the comparison within the context of Gilmore and Green's match-mismatch hypothesis, which specifies that problem-solving performance depends upon whether the structure of a problem is matched by the structure of a notation [46, 47]. Actually, Gilmore and Green's match-mismatch hypothesis is a somewhat stronger statement than the basic principle that the benefits of a notation are relative to a particular task. The match-mismatch hypothesis states that every notation highlights some kinds of information, while obscuring others. Thus, the match-mismatch hypothesis also predicts that every notation will perform relatively poorly for certain tasks.

Gilmore and Green's research had established a specific instance of this principle involving textual forms of conditional logic. In particular, Green had formerly identified two conditional notations: a "forward" notation which is based on the if-then-else statements common to procedural programming languages and a "backward" notation called the do-if statement which is based on production system notation (Figure 9). The match-mismatch which Gilmore and Green observed with these two textual forms of conditional logic is that the if-then-else notation facilitates answering forward questions (i.e., "which action results for given conditions?") while the do-if notation facilitates answering backward questions (i.e., "which conditions must exist to evoke a specified action?").

LabVIEW happens to provide two notations for expressing conditional logic. A gates notation uses the more familiar idiom of a circuit diagram (Figure 10). The boxes notation uses the more unusual method of showing only one logical path at a time (Figure 11); the user must toggle buttons to see the various paths through the conditional logic. Green and Petre felt that these two notations correspond to both a forward (the boxes notation) and a backward form (the gates notation) of conditional notation. To test this speculation, Green and Petre decided to study the two LabVIEW notations using forward and backward questions; if their hypothesis was correct, a match-mismatch

Forward Notation
(if-then-else)

```
if high:
   if wide:
      if deep: weep
      not deep:
         if tall: weep
         not tall: cluck
         end tall
      end deep
   not wide:
      if long:
         if thick: gasp
         not thick: roar
         end thick
      not long:
         if thick: sigh
         not thick: gasp
         end thick
      end long
   end wide
not high:
   if tall: burp
   not tall: hiccup
   end tall
end high
```

Backward Notation
(do-if)

```
howl:     if honest & tidy & (lazy | sluggish)
laugh:    if honest & tidy & ¬ lazy & ¬ sluggish
whisper:  if honest & ¬ tidy &
          (nasty & greedy | ¬ nasty & ¬ greedy)
bellow:   if honest & ¬ tidy & nasty & ¬ greedy
groan:    if honest & ¬ tidy & ¬ nasty & greedy
mutter:   if ¬ honest & sluggish
shout:    if ¬ honest & ¬ sluggish
```

Figure 9: Green's textual forward and backward representations for conditional logic. The if-then-else notation facilitates answering questions which require working from inputs (conditions) to outputs (actions). The do-if notation facilitates working from outputs to inputs. (Adapted from [9] by permission of T. R. G. Green).

would result. Plus, Green and Petre wanted to compare LabVIEW's performance to that of the two textual notations. They devised a study involving two parts. In Part 1, subjects were shown programs and were given one question (either a forward or backward question) per program. In Part 2, subjects were tested with "same-different" questions. In other words, subjects were given pairs of programs; for each pair, they were asked whether the two programs were functionally equivalent. The independent variables of this study were notation (visual versus textual), form of conditional expression (forward form versus backward form) and type of question (forward question versus backward question), all of which were within-subjects factors. The dependent variables were correctness and time. A total of 11 subjects took part, all of whom were experienced programmers. Furthermore, five subjects (Group 1) had used LabVIEW in their work for six or more months, and the remaining six (Group 2) were experienced in using electronics schematics.

Looking at the overall results, time was the more sensitive measure of the two dependent variables. In Part 1, there were significant effects for both aspects of the study. As for the match-

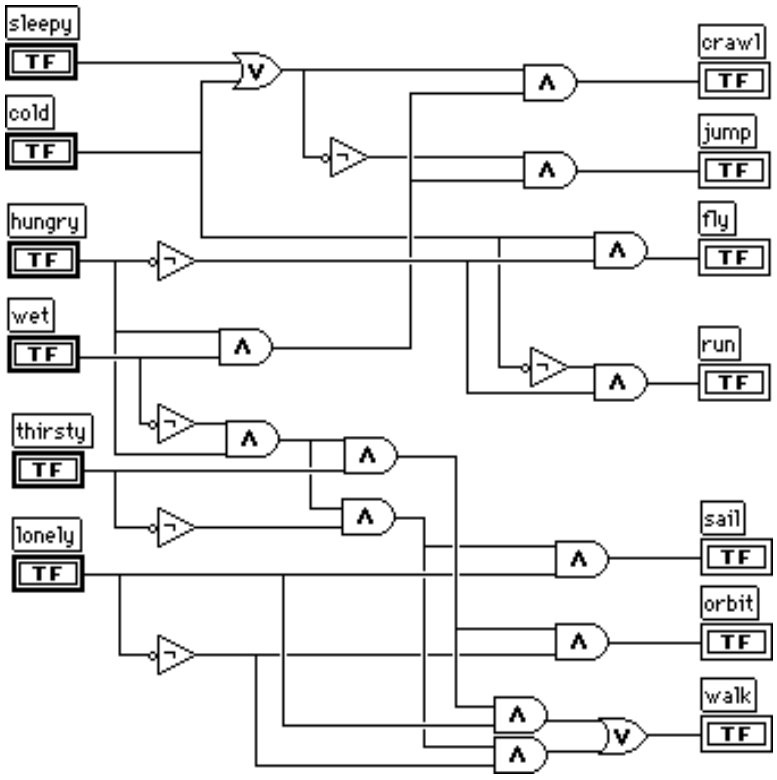

Figure 10: LabVIEW's gates representation for conditional logic. The gates perform the boolean operations AND, OR and NOT. Input for the conditional logic flows from the left to the right. (Reprinted from [9] by permission of T. R. G. Green).

mismatch hypothesis, the LabVIEW notations did exhibit the expected effects. As for the visual versus textual comparison, the visuals were slower than the text in all conditions. In fact, the text outperformed LabVIEW for each and every subject (each subjects' mean visual versus text times were compared). The overall mean times were 35.2 seconds for text compared to 68.1 seconds for LabVIEW. Analysis indicates that notation alone accounts for 25.0% of the total variance in the time data. In Part 2, Green and Petre did not find a significant match-mismatch behavior. They did find more evidence that the visual notation was causing longer response times.

One flaw with the study can be noted. Green and Petre note that, during Part 2, Group 1 subjects received only same-different comparisons in which the pairs were composed of one text program and one visual program. Thus, the full range of data necessary to contrast the performance of the visual versus textual representations was not available for the Group 1 subjects. This flaw was corrected for the Group 2 subjects. Additionally, some researchers raise a point of contention, citing the static nature of the stimulus programs used in this study as a second flaw. Specifically,

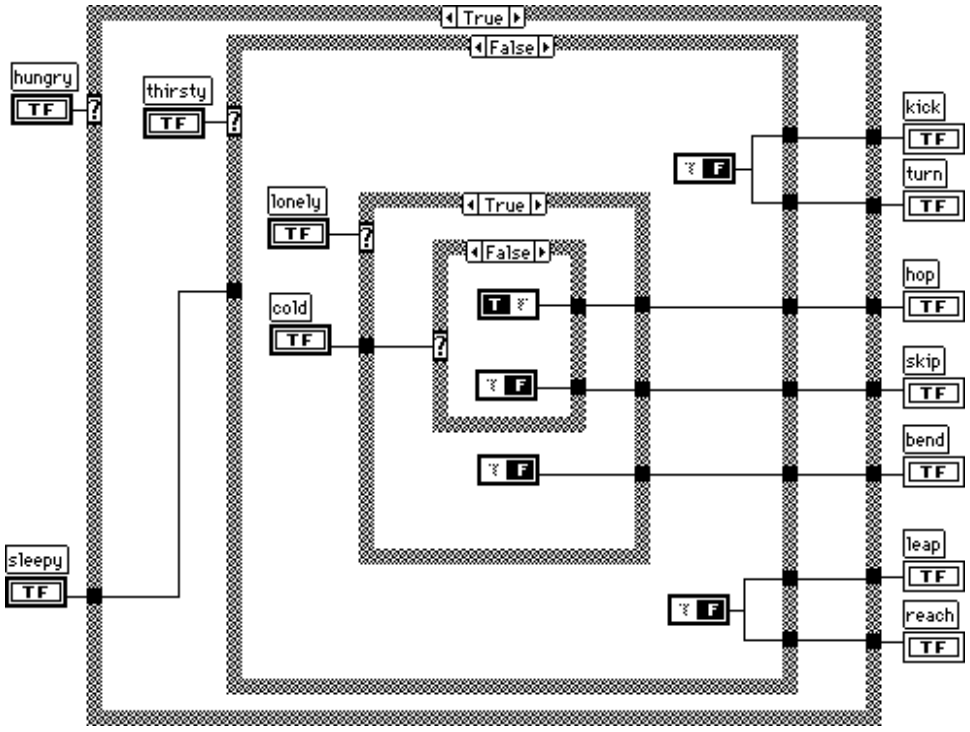

Figure 11: LabVIEW's boxes representation for conditional logic. This notation is interactive in a sense. To see the two paths of a true/false statement, the user clicks on the statement's true/false button to toggle between views of the true and false arms of the statement. (Reprinted from [9] by permission of T. R. G. Green).

Pandey and Burnett [7] and Baroth and Hartsough [31] believe that, because subjects could not fully interact with the stimulus programs, the study was biased against LabVIEW. This belief rests on the idea that benefits of VPLs stem from the responsive nature that is characteristic of some VPLs. In the Green and Petre study, it is not completely clear how this issue applies, given that the boxes notation was, by definition, interactive to a limited degree. In any case, the visual programming community needs controlled studies, such as Green and Petre's study, that separate the effects of the visual primitives of VPLs from additional interactive aspects of the VPLs' environments. In this way, researchers can learn whether the visual aspects of a language add any benefits that an interactive textual language cannot.

The Green and Petre study is quite illuminating. The match-mismatch hypothesis was upheld in this experiment. Yet, the magnitude of the match-mismatch effect was less than anticipated. Instead, Green and Petre were struck by the difficulties created by the visual notations especially given the experience level of the subjects and also given the fact that the LabVIEW programs had

been laid out according to accepted style conventions and critiqued by expert circuit designers. Indeed, the difficulty of the visual notations was the strongest effect observed in this study.

4.2.3 LabVIEW, petri nets and secondary notation

In subsequent articles, Green and Petre report additional observations from their LabVIEW study which raise two more topics [38, 39]. Specifically, they argue that "secondary notation" accounts for a large part of the (un)readability of a visual notation and, moreover, that the ability to read a visual notation and its associated secondary notation is dependent upon training. Secondary notation refers to layout of a (visual) program and includes techniques such as clustering and use of white space; one well known technique in textual programming is indentation. Green and Petre coined the term secondary notation for these perceptual aspects of good programming style in order to emphasize that these layout cues are not usually a formal part of a language's definition, yet are influential in determining the comprehensibility of programs written in the language.

Green and Petre make their argument using observational studies in which they watched expert electronics designers at work and from corroborating observations taken from the LabVIEW study. In their observational studies, they found indications that experts are distinguished from novices in their ability to use secondary notation. In the LabVIEW study, Green and Petre also made similar observations. Although there were no statistically significant performance differences between the Group 1 and the Group 2 subjects, Green and Petre observed behavior that led them to suspect a difference in the expertise levels of the two groups. Group 1 subjects displayed behavior that Green and Petre attribute to relative inexperience. As a group, the Group 2 subjects approached the study questions fairly consistently; the subjects tended to use like strategies and to choose their strategies based upon the style of the problem. In contrast, the Group 1 subjects were inconsistent in their strategies, even to the point of changing strategies in mid-task. Also, whereas Group 2 recognized and took advantage of sub-term groupings, Group 1 seemed unaware of this secondary notation. In sum, expertise could not overcome everything, and the LabVIEW notations were consistently slower than the text. Yet, the subjects' performance hint at the importance of secondary notation and expertise and the need for further study of these topics.

Interestingly, the Green and Petre results have been further explored by Moher et al. in a study that used the same experimental setup to compare text and petri nets [40]. Moher et al.

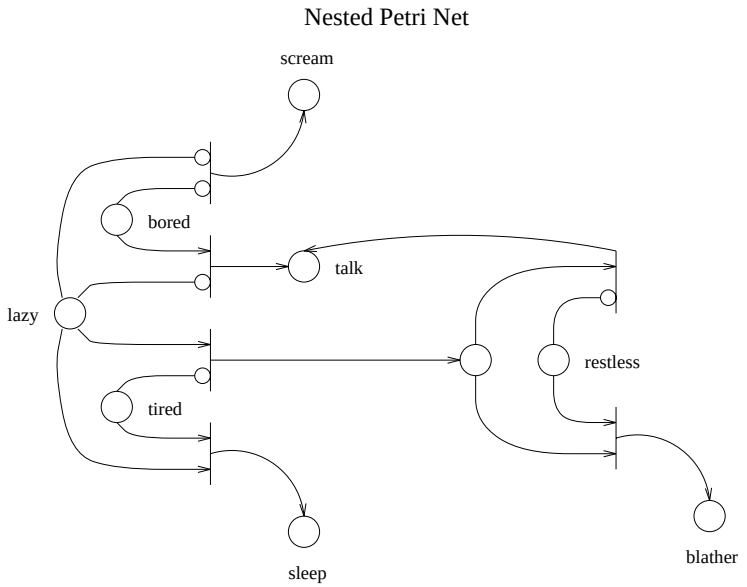

Figure 12: Moher et al.'s forward representation for conditional logic. Their petri net notations allow for inhibitor arcs which are indicated by small circles drawn in place of the arrowheads that normally terminate the arcs. (Adapted from [40], page 140, by permission of Ablex).

were interested in whether petri nets show promise as the visual basis for a VPL. They were also interested in comparing their results with the Green, Petre and Bellamy results [32]; the Green, Petre and Bellamy paper reports on preliminary results (the scores from the Group 1 subjects) of the later Green and Petre paper. To achieve both ends, Moher et al. used the same experimental design, the same software for testing and data collection and the same stimulus programs and questions employed in the Green, Petre and Bellamy study. Similarly, their independent and dependent variables were the same. Their subjects were faculty members or advanced graduate students; all were experienced in conditional logic and in the use of petri nets.

The difference between Moher et al. and the LabVIEW study was, of course, the choice of visual representation. Moher et al. designed three different petri net notations (Figures 12 and 13). They designed a nested net form which they felt corresponded to the if-then-else statement (a forward form); they designed distinguished net and articulated net forms as analogs to the gates notation and the do-if statement (backward forms) respectively. An important aspect of these notations is that, unlike any other study summarized in this paper, the same conditional logic expressed in the three petri net notations differs only in secondary notation. Whereas the visual representations contrasted in the other studies differed in their use of visual shapes (syntax) and

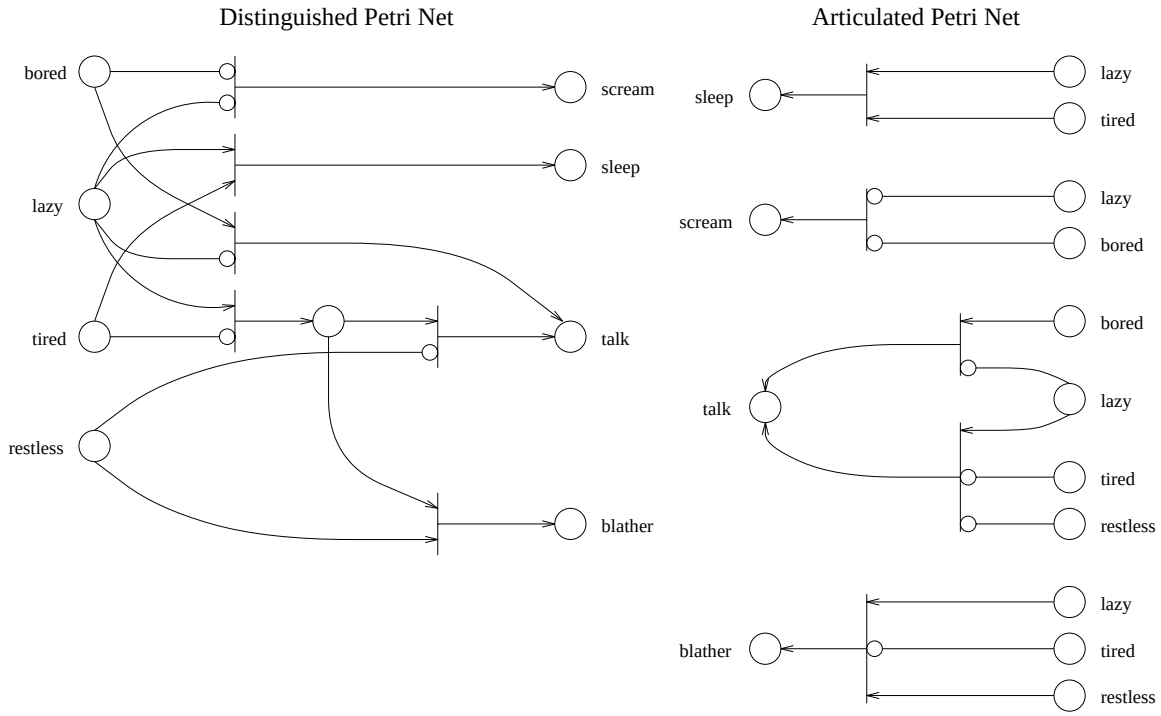

Figure 13: Moher et al.'s backward representations for conditional logic. In the distinguished petri net notation, conditions are arranged vertically on the left separately from the actions. The actions are grouped vertically on the right. In the articulated petri net notation, each action gets its own subnet. Conditions are arranged on the right and are repeated as needed throughout the subnets. (Adapted from [40], pages 142 and 143, by permission of Ablex).

in the semantics attributed to those shapes, the petri net notations differed only in how the same set of visual objects was arranged on the screen. Thus, Moher et al. became the first investigators to target directly in their experiment the effects of secondary notation in visual programming notations. Another notable aspect of these notations is the reasoning behind the articulated net design. Moher et al. observed that the do-if notation in the Green, Petre and Bellamy paper may have had an advantage. The do-if notation articulates each possible action separately by using repeated conditions; in contrast, the gates and distinguished net notations use an integrated format with nonrepeating conditions at the cost of overlapping lines. To counter this possible advantage, Moher et al. devised the articulated net notation.

The Moher et al. study yielded several significant results which were all manifested in the time data. In Part 1, the match-mismatch hypothesis was replicated for the textual notations but not for the petri nets. Rather, the petri nets were faster overall for backward questions than for

forward questions. In addition, the petri nets consistently performed worse than or equal to the text counterparts. The strongest case of a petri net notation equaling the text notations came from the articulated nets: The response times from use of the articulated nets were not significantly different from the do-if notation. Interestingly, the petri nets compared favorably with the scores achieved with the LabVIEW notations in the Green, Petre and Bellamy study; the Moher et al. subjects outperformed the LabVIEW Group 1 subjects by about 20%, even without considering the faster times of the articulated net. In Part 2, since Moher et al. used only pairs in which one program was in a text form while the other was in graphic form, they did not have the data necessary to make comparisons along the text/visual dimension. However, the Part 2 data did exhibit a significant match-mismatch for the pairs; response times were reduced when subjects dealt with 'matched' pairs (pairs in which the two programs were either both forward forms or backwards forms) compared to 'unmatched' pairs.

In assessing the Moher et al. study, note that the petri nets designed by Moher et al. have particular characteristics; Moher et al. represented conditional logic using deterministic petri nets which yield a single output token (action) from the input tokens (conditions). Consequently, this study did not probe the qualities normally associated with petri nets in their more common role as a modeling and specification tool for complex, concurrent and/or non-deterministic systems. Yet, the results shed light on two topics. First, the results concur with the Green and Petre results that the match-mismatch hypothesis cannot account for all of the differences seen in this experiment. Second, the results uncover a situation in which comprehension is dependent upon secondary notation. In most cases, the text outdid the petri nets. However, the articulated nets performed identically to the do-if statements. A related irony is that the nested nets, which is the layout style traditionally considered best for petri nets (which prescribes minimizing wire crossings), performed worst among the three petri net forms.

4.2.4 Algorithm animation as a pedagogical aid

A handful of empirical studies evaluating algorithm animation have been conducted, all of which focused on the use of animations as a pedagogical tool. Despite popular enthusiasm for the idea of algorithm animations, these studies have found no conclusive evidence to recommend their use. For example, Stasko, Badre and Lewis examined the effects of algorithm animation using

an XTango animation of a heap algorithm [42]. Stasko, Badre and Lewis studied students in a between-subjects design to measure learning via a textual description versus learning via text accompanied by an animation. Each student group was exposed to one of the two experimental conditions, after which both groups answered questions on a comprehension test. The independent variable was the presence of the animation, and the dependent variable was the correctness scores from the comprehension test. No significant difference occurred in the comprehension test scores. Similar results occurred in a subsequent study conducted by Lawrence, Badre and Stasko [41]. This followup experiment used a more complex design to evaluate the effects an algorithm animation when used in a classroom setting and in a manner that allows direct student involvement in the learning process. Once again, animation did not make a difference in comprehension test scores. Although, given that this study isolated the the static versus dynamic aspect of animation, the results are not relevant to the question of whether the chosen visual notation was beneficial. All of the groups were exposed to either an animation or a sequence of transparencies (snapshots from the animation); the contrast was not between a textual with a visual representation, as in the earlier study.

In view of this failure to empirically validate animation, Gurka and Citrin advocate a careful meta-analysis of the existing work, looking especially for factors that might have produced false negative results [43]. From their own meta-analysis, Gurka and Citrin have identified 10 experimental design issues that may account for previous disappointing results. Three of the factors are general experimental issues, while the remaining seven pinpoint specific aspects of animation systems that may confound a study's results. Researchers conducting further empirical study should consider these issues when designing their studies. Alternatively, the algorithm animation community must begin to consider that the repeated negative results mean that animation yields no improvement over standard teaching techniques.

4.2.5 Spreadsheets and the problem of visibility

Arguably, the most successful VPL, possibly the most successful programming language in general, has been the spreadsheet. Due to its success, the spreadsheet has captured the attention of some language designers who wish to reuse its successful features in other programming languages. Correspondingly, several papers have been written about the strengths of the spreadsheet model.

However, the literature has largely been composed of expert opinion.

Of the existing empirical studies, an observational study by Hendry and Green stands out for its premise that language designers need to shift focus to the weaknesses of spreadsheets [22, 44]. Hendry and Green stress that any programming language design is a point in a design space whose position represents the many decisions made during the design process. Usually, the decisions attempt to facilitate a certain kind of programming; often, decisions favoring certain cognitive tasks result in tradeoffs against other desirable cognitive aspects. Hendry and Green argue that overlooking the weaknesses of spreadsheets fosters too shallow an understanding of spreadsheets, in turn blocking an avenue for improvement.

Hendry and Green's observational study consisted of interviews with 10 spreadsheet users. The subjects were end users who used computers as a part of their everyday work. In the interviews, the subjects were asked about their experiences with creating and comprehending spreadsheets; they were also asked to explain how one of their own spreadsheets worked. The subjects and the spreadsheets that they explained cover a wide range of spreadsheet use: the subjects worked in a variety of work environments and domains, and their spreadsheets ranged in size (from 1 to 14 pages in length), in complexity (from using solely simple formulas to using formulas containing as many as 15 cell references), in subject matter (from financial reporting to complex models of global warming) and in time to create (some had taken only a few hours to create, while others had been developed over several years).

In the interviews, the subjects echoed commonly-held beliefs concerning the benefits of spreadsheets, especially the idea that the spreadsheet paradigm supports quick and easy setup of spreadsheets. Yet, the interviews also brought out difficulties that the subjects faced in using spreadsheets, sometimes stemming from surprisingly simple-sounding tasks. One of the larger problems with current spreadsheets is poor visibility. This refers to the manner in which formulas and cell references are dispersed throughout spreadsheet cells. In effect, the norm for most spreadsheet packages is that the formulas are hidden throughout the interface. The visibility problem creates difficulties for users in locating cell dependencies, in building a global vision of the spreadsheet and in debugging.

Hendry and Green's findings regarding poor visibility are consistent with observations made in an earlier study conducted by Brown and Gould [45]. In their study, Brown and Gould observed nine

experienced spreadsheet users as each created three spreadsheets. Brown and Gould found that, even though their subjects were "quite confident" that their spreadsheets contained no errors, 44% of the spreadsheets contained errors. The majority of the errors occurred in formulas and included mistakes such as incorrect cell references. As a result of both sets of observations, Hendry, Green, Brown and Gould propose that spreadsheet visibility/debugging tools would be an improvement to the standard spreadsheet interface.

In short, Hendry and Green's study is valuable for two reasons. First, their study provides a list of specific weaknesses that warrant further attention. Second, beyond these particulars, Hendry and Green's study underscores the value of empirical study as a means of improvement. Notwithstanding the success of spreadsheets, Hendry and Green argue that we should consider their weaknesses, a major reason being that increased understanding can lead to improved language designs.

4.3 Summary of the Evidence Against

The efficacy of a notation depends on the task to be performed. This point was illustrated in Section 3. Here in Section 4, it was raised again as a slightly stronger principle called the match-mismatch hypothesis. Both the Green and Petre study and the Moher et al. study support this hypothesis. Although not as strong a result, the Wright and Reid study also hints at the importance of considering the effects of a notation over a range of tasks. The question for VPL research becomes, "Given the range of information required in programming, can a VPL highlight enough of the important information to be of practical benefit?"

The Hendry and Green observational study of spreadsheets also underscores the lesson of the match-mismatch hypothesis. The marketplace success of the spreadsheet makes it clear that the spreadsheet is useful; yet, even so, the spreadsheet does not facilitate all cognitive tasks. In addition, Hendry and Green remind us that language designs are situated in a complex design space and that decisions made during the design process can unwittingly impede certain cognitive tasks. Empirical study can help shed light in such situations, thus acting as a means of improvement.

Some members of the VPL community dismiss the studies of flowcharts as irrelevant to current-day VPLs, their argument being that most current VPLs are not based on flowcharts. However, the flowchart studies should not be completely ignored. Their obvious applicability is to the VPLs that use visual representations to illustrate control flow. Thus, the flowchart studies have a direct

bearing on VPLs based on flowcharts, such as Pict [48] and FPL [30]. Moreover, The flowchart studies may also be germane to VPLs that, although not based on flowcharts per se, are designed to illustrate the control flow of a program; this category includes tree-structured notations [49, 50] and more exotic designs such as VIPR [51]. The implication of the flowchart studies is that control flow alone does not seem to comprise a large enough part of what makes programming difficult.

Of note from the Curtis et al. flowchart study is the strong role that individual differences played in that experiment. In their results, individual differences accounted typically for one-third to one-half of the variability seen in the dependent variables. As mentioned above, the impact of individual differences is a recurring theme in empirical studies in programming and one that might make the effects of VPLs difficult to pinpoint. This issue also leads one to remember the distinction between statistical significance and realized benefit (i.e., the magnitude of an impact). More important than the issue of whether a VPL has a statistically significant effect is the question of whether the effect size is large enough to be of practical interest.

Of note from the Ramsey, Atwood and Van Doren study, the use of space-saving techniques brings up the topic of the Deutsch Limit. The term Deutsch Limit refers to the relatively low screen density of visual notations compared to textual notations. This has long been a recognized issue in the visual programming community, one which fuels speculation that VPLs will fail to be practical. This issue involves not only how visuals affect the nature of text used in conjunction with the visuals, but also whether VPLs allow enough visual elements on a screen at one time.

Secondary notation is an issue for both textual and visual notations. In the case of text, the range of secondary notation techniques is relatively limited due to text's symbolic nature and its basic serial layout. In visual notations, secondary notation can potentially be much richer. If Green and Petre's assessments are correct, novice users will not immediately master the secondary notation of VPLs. It remains to be seen whether novice users have equivalent or greater problems with the secondary notation in visual as opposed to textual programs. This issue might be especially troublesome for VPLs aimed at end users.

Some of the difficulty in the studies of VPLs to date has stemmed from experimental design problems. The reasoned identification of the independent variable, careful choice of a valid dependent variable and clear notion of the task of the experiment are always important to the success

of a study. Yet, particular attention should be paid to these decisions in the relative haze of a poorly-understood area. So far in the VPL studies, the most common dependent variables have been time and correctness; Scanlan thinks that comprehension time is the most sensitive of the effects of visual notations.

The algorithm animation studies, as a group, have uncovered no benefits of animations for learning algorithms. This might truly reflect the fact that animations do not offer benefits over other teaching techniques. In contrast, Gurka and Citrin feel that this conclusion is premature. They present a meta-analysis of the existing animation studies that emphasizes the possibility that prior studies missed the effects of the animations. For example, in both the Stasko, Badre, Lewis study and the Lawrence, Badre and Stasko study, the measures of effectiveness were comprehension test scores. Maybe the effects of animation could be seen in a study that examined the time needed to learn an algorithm.

5 Conclusions

As this collection of results shows, visual representation can improve human performance. However, the fact remains that the VPL community lacks a well-founded cognitive or empirical argument in support of VPLs. There are two strategies for action. One course of action would be to continue developing VPLs, under the assumption that VPL designs will naturally evolve in appropriate directions and that users will recognize their worth. This strategy has worked for spreadsheets; spreadsheets are wildly successful, despite the shortage of validation studies. The second strategy would be to devote resources to empirical testing. After a decade of research, the VPL community now has a good handful of VPLs that are starting points for study. Solid evidence may be the crucial factor for the continuation of VPL research, an area which has met with some sharp criticism. Moreover, within the current context of programming, in which software engineering is struggling to move from being a craft to being an actual engineering discipline, scientific method beckons.

Acknowledgments

My gratitude goes to Doug Fisher, Margaret Burnett, Laura Novick and Jonathan Zweig, all of whom carefully reviewed drafts of this paper. My appreciation also goes to S. K. Chang and three anonymous referees. Thank you all for your constructive comments.

References

[1] M. Burnett, M. Baker, P. Bohus, P. Carlson, S. Yang, and P. van Zee. The scaling up problem for visual programming languages. *IEEE Computer*, 28(3):45–54, March 1995.

[2] K. Whitley. Visual programming languages and the empirical evidence for and against. Technical Report CS-96-04, Department of Computer Science, Vanderbilt University, Nashville, TN 37235, October 1996.

[3] N. Fenton, S. L. Pfleeger, and R. L. Glass. Science and substance: A challenge to software engineers. *IEEE Software*, 11(7):86–95, July 1994.

[4] R. L. Glass. The software-research crisis. *IEEE Software*, 11(11):42–47, November 1994.

[5] N. Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, SE-20(3):199–206, March 1994.

[6] T. R. G. Green. Noddy's guide to visual programming. *Interfaces*, 1995. The British Computer Society Human-Computer Group.

[7] R. K. Pandey and M. M. Burnett. Is it easier to write matrix manipulation programs visually or textually? An empirical study. *Proc. 1993 IEEE Symposium on Visual Languages (VL'93)*, pages 344–351, 1993.

[8] B. A. Price, R. M. Baecker, and I. S. Small. A principled taxonomy of software visualization. *J. Visual Languages and Computing*, 4:211–266, 1993.

[9] T. R. G. Green and M. Petre. When visual programs are harder to read than textual programs. *Proc. Sixth European Conference on Cognitive Ergonomics (ECCE 6)*, pages 167–180, 1992.

[10] I. Vessey and R. Weber. Structured tools and conditional logic: An empirical investigation. *Communications of the ACM*, 29(1):48–57, January 1986.

[11] A. F. Blackwell. Metacognitive theories of visual programming: What do we think we are doing? *Proc. 1996 IEEE Symposium on Visual Languages (VL'96)*, pages 240–246, September 1996.

[12] F. P. Brooks. No silver bullet: Essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, April 1987. Also appeared in H.-J. Kugler, editor, *Information Processing '86*, pages 1,069-1,076, Elsevier Science Publishers B. V., 1986.

[13] I. Vessey and R. Weber. Research on structured programming: An empiricist's evaluation. *IEEE Transactions on Software Engineering*, SE-10(4):397–407, July 1984.

[14] B. A. Sheil. The psychological study of programming. *ACM Computing Surveys*, 13(1):101–120, March 1981.

[15] R. E. Brooks. Studying programmer behavior experimentally: The problems of proper methodology. *Communications of the ACM*, 23(4):207–213, April 1980.

[16] J.-M. Hoc, T. R. G. Green, R. Samurçay, and D. J. Gilmore, editors. *Psychology of Programming*. Academic Press, London, 1990.

[17] T. M. Moran. Guest editor's introduction: An applied psychology of the user. *ACM Computing Surveys*, 13(1):1–11, March 1981.

[18] J. H. Larkin and H. A. Simon. Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, 11:65–99, 1987.

[19] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: A cognitive dimension framework. *J. Visual Languages and Computing*, 7(2):131–174, 1996.

[20] T. R. G. Green. Cognitive dimensions of notations. *People and Computers V: Proc. British Computer Society HCI'89*, pages 443–460, September 1989.

[21] F. M. Modugno, T. R. G. Green, and B. Myers. Visual programming in a visual domain: A case study of cognitive dimensions. *People and Computers IX: Proc. British Computer Society HCI'94*, pages 91–108, 1994.

[22] D. G. Hendry and T. R. G. Green. Creating, comprehending and explaining spreadsheets: A cognitive interpretation of what discretionary users think of the spreadsheet model. *Int. J. Human-Computer Studies*, 40(6):1033–1065, 1994.

[23] R. S. Day. Alternative representations. In G. H. Bower, editor, *The Psychology of Learning and Motivation*, volume 22, pages 261–305. Academic Press, 1988.

[24] J. M. Polich and S. H. Schwartz. The effect of problem size on representation in deductive problem solving. *Memory & Cognition*, 2:683–686, 1974.

[25] S. H. Schwartz and D. L. Fattaleh. Representation in deductive problem solving: The matrix. *J. Experimental Psychology*, 95:343–348, 1972.

[26] S. H. Schwartz. Modes of representation and problem solving: Well evolved is half solved. *J. Experimental Psychology*, 91(2):347–350, 1971.

[27] J. M. Carroll, J. C. Thomas, and A. Malhotra. Presentation and representation in design problem solving. *British J. of Psychology*, 71:143–153, 1980.

[28] C. McGuinness. Problem representation: The effects of spatial arrays. *Memory & Cognition*, 14(3):270–280, 1986.

[29] D. A. Scanlan. Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, 6(5):28–36, June 1989.

[30] N. Cunniff and R. P. Taylor. Graphical vs. textual representation: An empirical study of novices' program comprehension. *Empirical Studies of Programmers: Second Workshop*, pages 114–131, 1987.

[31] E. Baroth and C. Hartsough. Visual programming in the real world. In M. Burnett, A. Goldberg, and T. Lewis, editors, *Visual Object-Oriented Programming: Concepts and Environments*, chapter 2, pages 21–42. Manning Publications Co., 1995.

[32] T. R. G. Green, M. Petre, and R. K. E. Bellamy. Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture. *Empirical Studies of Programmers: Fourth Workshop*, pages 121–146, 1991.

[33] D. R. Raymond. Characterizing visual languages. *Proc. 1991 IEEE Workshop on Visual Languages (VL'91)*, pages 176–182, 1991.

[34] P. Wright and F. Reid. Some alternatives to prose for expressing the outcomes of complex contingencies. *J. Applied Psychology*, 57(2):160–166, 1973.

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

[36] H. R. Ramsey, M. E. Atwood, and J. R. Van Doren. Flowcharts versus program design languages: An experimental comparison. *Communications of the ACM*, 26(6):445–449, June 1983.

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

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

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

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

[41] A. Lawrence, A. Badre, and J. Stasko. Empirically evaluating the use of animations to teach algorithms. *Proc. 1994 IEEE Symposium on Visual Languages (VL'94)*, pages 48–54, October 1994.

[42] J. T. Stasko, A. M. Badre, and C. Lewis. Do algorithm animations assist learning? An empirical study and analysis. *Proc. INTERCHI '93 Conference on Human Factors in Computing Systems*, pages 61–66, April 1993.

[43] J. S. Gurka and W. Citrin. Testing effectiveness of algorithm animation. *Proc. 1996 IEEE Symposium on Visual Languages (VL'96)*, pages 182–189, September 1996.

[44] D. G. Hendry and T. R. G. Green. Cogmap: A visual description language for spreadsheets. *J. Visual Languages and Computing*, 4(1):35–54, 1993.

[45] P. S. Brown and J. D. Gould. An experimental study of people creating spreadsheets. *ACM Transactions on Office Information Systems*, 5(3):258–272, July 1987.

[46] D. J. Gilmore and T. R. G. Green. Comprehension and recall of miniature programs. *Int. J. Man-Machine Studies*, 21:31–48, 1984.

[47] T. R. G. Green. Conditional program statements and their comprehensibility to professional programmers. *J. Occupational Psychology*, 50:93–109, 1977.

[48] E. P. Glinert, M. E. Kopache, and D. W. McIntyre. Exploring the general-purpose visual alternative. *J. Visual Languages and Computing*, 1:3–39, 1990.

[49] M. Aoyama, K. Miyamoto, N. Murakami, H. Nagano, and Y. Oki. Design specification in Japan: Tree-structured charts. *IEEE Software*, 6(3):31–37, March 1989.

[50] W. K. McHenry. R-Technology: A Soviet visual programming environment. *J. Visual Languages and Computing*, 1:199–212, 1990.

[51] W. Citrin, M. Doherty, and B. Zorn. Formal semantics of control in a completely visual programming language. *Proc. 1994 IEEE Symposium on Visual Languages (VL'94)*, pages 208–215, October 1994.