Are visual programming languages better? The role of imagery in program comprehension

Article *in* International Journal of Human-Computer Studies · June 2001

DOI: 10.1006/ijhc.2000.0465 · Source: DBLP

Raquel Navarro-Prieto
36 PUBLICATIONS **543** CITATIONS
SEE PROFILE

José J. Cañas
University of Granada
190 PUBLICATIONS **3,546** CITATIONS
SEE PROFILE

Int. J. Human-Computer Studies (2001) **54,** 799–829
doi:10.1006/ijhc.2000.0465
Available online at http://www.idealibrary.com on IDEAL®

Are visual programming languages better? The role of imagery in program comprehension

RAQUEL NAVARRO-PRIETO† AND JOSE J. CAÑAS
Departamento de Psicología Experimental, Facultad de Psicología, Universidad de Granada, Spain. email: raquel.navarro-prieto@motorola.com

(Received 8 October 1999, and accepted in revised form 10 October 2000)

This paper presents one experiment to explain why and under which circumstances visual programming languages would be easier to understand than textual programming languages. Towards this goal we bring together research from psychology of programming and image processing. According to current theories of imagery processing imagery facilitates a quicker access to semantic information. Thus, visual programming languages should allow for quicker construction of a mental representation based on data flow relationships of a program than procedural languages. To test this hypothesis the mental models of C and spreadsheet programmers were assessed in different program comprehension situations. The results showed that spreadsheet programmers developed data flow based mental representations in all situations, while C programmers seemed to access first a control flow and then data flow based mental representations. These results could help to expand theories of mental models from psychology of programming to account for the effect of imagery.

KEYWORDS: imagery; mental model; visual programming language; spreadsheets; C; program comprehension.

Program comprehension is an important part of computer programming skill, both from a practical and a theoretical perspective. It is a complex cognitive skill, which involves the acquisition of a mental representation of program structure and function. From a theoretical viewpoint, comprehension involves the assignment of meaning to a particular program, an accomplishment that requires the extensive application of specialized knowledge. Thus, research on program comprehension provides an effective means for studying the role of particular kinds of knowledge from a cognitive psychology perspective.

From a practical viewpoint, the ability to understand programs written by others is an important component of a programmer's skill set. During program comprehension the programmer finds her/himself performing programming tasks such as debugging, modification and, in general, code reuse. Pennington (1987*a*) estimated that more than 50% of all professional programmer time is spent on program maintenance tasks that involve modification and updates of previously written programs. Because other programmers

†Author to whom correspondence should be addressed. Human–Device Interaction Researcher, Motorola UK Research Lab, Viables Industrial Estate, Jays Close, Basingstoke RG22 4PD, UK.

most often write the programs, comprehension plays a central role in this endeavour. Therefore, research on effective comprehension strategies may provide information that can be useful to improve programmer performance, education, design technologies and programming environments.

This research is especially important for visual programming languages (VPLs) because during the past decade there has been a rapid emergence of these languages in private industry, based on the intuition that VPLs enhance programming. However, researchers have constantly pointed out the lack of evidence, as well as the lack of a theory that explains the effects of VPLs (Whitley, 1997). Golin (1991) remarked that "VPLs have largely been approached using *ad hoc* techniques. Visual languages are often specified intuitively, by giving examples of programs or informal descriptions of the structure of a visual program." To this day, the situation remains the same (Whitley, 1997).

Visual programming languages (VPLs) developers claim that these languages facilitate program comprehension because of their use of visual information. However, as Cunniff and Taylor (1987) pointed out, "there is little conclusive empirical research either to support or disprove the intuition that graphical representation is superior to text". In contrast with the amount of research that has been done in textual programming languages comprehension, there have been very few attempts to develop a theory about how VPLs are comprehended. The lack of empirical research is more visible for spreadsheets, especially if we take into account the number of users. As Hendry and Green (1994, p. 1034) pointed out "Yet its literature is tiny. Nardi and Miller (1990) remarked on the thinness of literature on spreadsheets, and the situation has changed little."

It has been claimed that the most powerful feature of spreadsheets is that people can see what is to be done and simply do it (Kay, 1984). Is that really true? What does that mean in terms of the concrete advantage of spreadsheets? Our research attempts to address the issue that would allow us to expand text comprehension theories to explain how VPL programmers develop *mental representations* of their programs.

Therefore, in what follows we present research aimed to find empirical data and theoretical explanations for the assumed advantage of VPLs over textual languages. Our proposal is based on the integration of data and methodology from Psychology of Programming and Image Processing research. We claim that Basic Psychology research concerning image processing can provide us a theoretical framework to interpret and explain the imagery effects in the area of program comprehension. First, we will start by reviewing what has been done in the area of program comprehension and visual programming languages. Second, we will review the Imagery effects, which have been found in image processing, human computer interaction and computer assisted instruction. Then, we will present a summary of all these data and the hypothesis derived from them. The rest of the paper will be dedicated to the experiment conducted to test this hypothesis and the conclusions derived from our research.

1. How are programs comprehended?

1.1. EARLY WORK: HIERARCHICAL PLAN MODELS

Early research on program comprehension was based mainly on the so-called Plan Theories of programming and consisted of studying the stereotypical knowledge stored in long term (plans, schemas) which the programmers would use when trying to understand a program. The main assumption of plan-based theories was that the development of expertise is associated with the gradual accumulation of plan knowledge over time. The first approaches, like Shneiderman (1976) and Soloway and Ehrlich (1984), were based on Schema theories (Schank & Abelson, 1977), and the studies of chess expertise (DeGroot, 1965). These models proposed that program understanding is a top-down process in which experts access hierarchically organized plan knowledge. Therefore, when programmers are trying to understand a program they first look for the plan that best matches this program. Once the programmer has chosen the plan that corresponds to the program she/he is reading, this plan guides the rest of the program comprehension process. More recently, Rist (1986) proposed a more general model in which this top-down process is combined with a bottom-up process. However, in his model schema knowledge still plays a leading role in program comprehension.

In general, the studies conducted by the proponents of these theories supported the idea that experts possess structured long term memory (LTM) knowledge. However, later studies failed to reproduce the earlier results or to prove the predictions from the theories (Widowsky & Eyferth, 1986; Vessey & Weber, 1984), suggesting that the existence of plans is not enough to explain the complex task of program understanding. In particular, studies in reading behaviour such as that of Robertson, Davis, Okabe and Fitz-Randolf (1990) pointed out that eye movements through the code give support to Rist's model of nonlinear understanding, but also showed evidence of strategic search. Also, some empirical results showed differences between languages in terms of the types of errors committed by the subjects (Gilmore & Green, 1988) which show the necessity of taking into account language variables. These language variables could not be explained in terms of general, abstract plan knowledge.

1.2. PROGRAMMING STRATEGY MODELS

As an alternative to these knowledge-based theories it was suggested that expertise in programming might involve the development or acquisition of complex task-specific cognitive problem-solving strategies. Then, the focus of the researchers moved to the study of the strategies that programmers use to determine which knowledge should be appropriate in a concrete situation.

One of the most representative researchers from this perspective was Davies who proposed the "Knowledge Restructuring Model" (Davies, 1991, 1994). The model was based on previous work concerned with other complex problem-solving tasks which suggested several cognitive mechanisms to explain the development of expertise and its relationship with knowledge structure and organization (Rumelhart & Norman, 1978). The principal assumption of Davies' model was that knowledge units are restructured at the level of individual schema, rather than at a global level common to all programmers. He suggested that at later stages of skill acquisition, particularly during the transition from intermediate to expert, the global restructuring of knowledge becomes less important than the internal structuring of individual schema.

The key component of this model, contrary to the plan theories, was that the restructuring process is related to the strategies used by the programmers in a situation. Therefore, the difference between expert and intermediate programmers is that the former apply adequate strategies which allow to optimize the restructuring process itself and the use of these schemas in programming. By contrast intermediate programmers possess the plan knowledge but cannot restructure this knowledge because of the lack of the needed strategies. For instance, one of these strategies is the one that allows the experts to optimize the use of the information provided for the external program representation about the control flow and data flow of the program (Davies, 1991).

Since these models moved the focus of the attention from the long-term stored plans to more strategic aspects, paying special attention to characteristics of the specific programming situation, could easily explain how task's variables (e.g. the programming language) would interact with subjects' variables (e.g. the level of experience). In this regard, Davies' model was able to explain how at certain skill levels the notation would have different effects (Davies, 1994).

Although strategy models were a step forward by pointing out the role of the strategies, they did not explain in detail how the knowledge about structures and strategies is combined in a concrete situation to develop a mental representation in the working memory, during program comprehension. That was the goal of the "Mental Model" theories of program comprehension.

1.3. MENTAL MODEL APPROACH

In this approach, in general, researchers regard a computer program as a text that gives some instructions to the computer to achieve a goal. The main goal of the text comprehension models of programming is to explain how this knowledge in conjunction with the program text and task characteristics contributes to the representation of the program (Schmalhofer & Glavanov, 1986). The research adopted the dominant point of view according to which various knowledge structures relevant to the text are activated in the working memory, in the course of comprehension of the text (Rumelhart, 1980).

The researcher that best represents this approach in programming is Pennington (1987*a*, *b*). In her work, she explored the role of two kinds of knowledge that programmers have about programming: text structure knowledge (TS knowledge) and knowledge of program plans (PK knowledge).

Pennington defined *text structure knowledge* as the programmer's knowledge of the control flow structure of a program text. Control flow representation is structured in terms of the sequence in which program actions will occur.

The second kind of knowledge, *plan knowledge*, emphasizes programmers' understanding of which patterns of program instructions go together to accomplish certain functions (Soloway, Ehrlich & Black, 1983; Soloway, Ehrlich & Bonar, 1982). Pennington suggested that plan knowledge could also guide the formation of a comprehension macrostructure.

Pennington tried to find evidence for the roles of text structure (TS) knowledge and plan knowledge (PK), and to determine which of these two views better characterizes program comprehension. In one experiment, she used a priming recognition technique,† and questions about several aspects of the programs to examine relations between

†A detailed description of this technique will be provided in pages 818 and 819.

program statements in program text representation. The recognition data showed, as predicted by a TS analysis of program comprehension, that only the target preceded by targets related by TS showed quicker response times. The questions about detailed operations and control flow relations were answered more accurately, while more errors were made on data flow, state, and function items. Another interesting result was that FORTRAN programmers made fewer recognition errors compared to COBOL subjects. Patterns of error rates for information question categories also differed for the two languages. The most noticeable difference between FORTRAN and COBOL patterns was the elevated error rate on function questions for COBOL subjects, while the data flow questions showed a slightly higher accuracy for COBOL programmers. Based on further analysis of these data, Pennington concluded that the pattern showed by FORTRAN programmers represents a later stage of comprehension process. The results from the second experiment, where programmers studied and modified a program of moderate length (200 lines), gave support to findings from the first experiment.

Pennington explained the results from both experiments in terms of the situational model proposed by the van Dijk and Kintschs' (1983) Text Comprehension Theory. While the textbase representation includes the micro and macrostructure of the program, the situational representation is a mental model of what the text is about referentially. To summarize her conclusions, she proposed that programmers go through two phases when they try to understand a program. In the first phase, they segment the program on the basis of control flow relationships (e.g. in control patterns like loops or conditional patterns). In the second phase, they try to understand the major program functions and data flow relationships. In performing a relatively easy task, like reading a short program, the first phase is all they use. However, to modify the program they will go through the second phase, understanding the data flow structure.

In a second paper, Pennington (1987*b*) addressed the question of how programmers who attain different levels of program comprehension differ in terms of their comprehension strategies and the resulting mental representations of the program text. For this goal she analysed the data from the summaries provided by the subjects in the experiment reported before, and related it with the answers to the comprehension questions. These analyses showed that top comprehenders' summaries included coordination of levels of representation program and domain, a procedural orientation for program level statements, and lower error rates across all categories. In contrast, poorer comprehenders expressed their summaries in terms of either the program model or the domain model but not both. This was accompanied by very high error rates for connections to the domain world and functions.

As a conclusion, from the differences found between the textbase model and the situational model, Pennington defined these two representations as: (1) a representation that highlights procedural program relations in the language of programs, called the *program model*, which corresponds to a textbase representation; (2) the *domain model*, which highlights functional relations between program parts that are expressed in the language of domain world objects corresponding to a situational model. She proposed that these representations could be considered, together, part of the programmers' mental model. In the words of the author "the essential message of this research is that to understand skilled intellectual performance we have to study cognitive representations of the stimulus. Furthermore, we must be prepared for a multiplicity of mental representations, even within one head. In the case of programming, at least two mental representations, are needed, a model of the program text and a model of the domain to which it refers, in order to understand the behaviour of experts". (Pennington, 1987*b*, p. 112).

1.4. SUMMARY

As a consequence of the empirical evidence in favour of mental model theories, research is now focused on how programmers develop several types of mental representation in program comprehension more than on how the information is stored in long-term memory. Thanks to the application of the theory of text comprehension (van Dijk & Kinstch, 1983) it is now possible to propose very specific hypotheses that guide systematic research and allow us to formulate explicit theories about the contents of the mental models and the variables that affect their formation.

Data from several studies seem to support Pennington's two stage theory and have cleared some of the variables that affect mental model developing in program comprehension. Some data confirmed that the length of the program and the level of experience (Pennington, 1987*a*; Corritore & Wiedenbeck, 1991) affect the process of developing these representations. Research in the last few years has shown the importance of systematic experiments that could explain the effect of different aspects on these representations, such as the purpose of reading and the language characteristics (Burkhardt, Détienne & Wiedenbeck, 1997; Corritore & Wiedenbeck, 1999; Wiedenbeck & Ramalingam, 1999).

2. Visual languages

The term "Visual Languages" has been used to refer to programming languages at the highest level of abstraction and include all systems that utilize pictorial information. They had been divided into two key branches: visual programming languages and program visualization systems (Myers, 1990; Kipper, Howard & Ames, 1997). Basically, in VPLs, the visual components are machine-processible primitives of the language. In contrast, within Visualization Systems, visual components are presented to the human user as a way of increasing human comprehension, not as executable commands. In the following sections, we will review the research in these two main areas: program visualization and visual programming languages. We will pay special attention to the explanations proposed and the general pattern, which could connect all these results. In both areas, the programming task to be performed seems to be a critical variable to explain their effectiveness.

2.1. PROGRAM VISUALIZATION

Program visualization systems (PVSs) are based on the assumption that execution of a computer task would be enhanced by providing users with a model of the interaction with the system. Many program visualization systems have been proposed for very specialized contexts, normally a specific application in industry. However, not all of them have had empirical support.

Researchers in this area have investigated the conditions under which and why the provision of a visual model of the system would facilitate the development of an adequate mental model of the program. In one of the first works in this area Mayer (1976) tested a model of BASIC. Two groups of students learnt to program in BASIC, one with the model and another without it. His data showed that the group that learnt without the model performed as well as the model group on problems that were similar to the learned material. However, on problems that required moderate amounts of transfer the model group excelled. Several further experiments have supported this explanation (Mayer, 1976, 1980; Mayer & Bromage, 1980) showing that the model was also more useful when material was poorly structured. These results were interpreted as evidence that the model helped the reader in holding information together, making the connection between the new technical information and a familiar analogy. Therefore, it facilitates transfer because it requires an understanding of conceptual ideas. The model does not facilitate retention because it only requires recall of specific code.

Other researchers have found evidence consistent with Mayer's data. Shih and Alessi (1994) investigated transfer between two skills involved in programming, code generation and code evaluation. Their data showed that a graphical model facilitated transfer from one skill to another, but only for the more difficult problems. Also, the group that learned with the model scored better in the mental model measures of comprehension and conceptual learning. However, in this case, as in most of the research in this area, the programmers' mental model was not directly measured.

The work of Cañas, Bajo and Gonzalvo (1994) is an interesting exception in this area because they directly measured subjects' mental models in programming learning. These authors investigated the effect of a tracer for the C programming language that visualized the internal state of computer memory and the sequence of computer actions during program execution. Using a Multidimensional Scaling Analysis technique, they found that the group that learned with the tracer, showed a greater weight in the semantic relationship dimension, while the students without the tracer showed more weight in the syntactic dimensions.

To summarize the research in this area, we conclude that the effect of pictorial aids in program visualization depends upon the programming task and the problem to be solved. They seem to be effective in enhancing program learning, especially for problems or skills in which the programmers were not trained. The data seem to indicate that pictorial aids improve a programmer's mental model, emphasizing the semantic relationships. These semantic relationships constitute the information that would facilitate better performance in problems and tasks for which the programmer was not trained.

2.2. VISUAL PROGRAMMING LANGUAGES

Visual programming languages (VPLs) consist of systems in which icons, symbols, charts and forms that are used to specify a program. We can differentiate three main categories of VPLs: languages with flowchart representations, languages with non-flowchart representation (e.g. LabView) and spreadsheet languages. In spite of the rapid emergence of these languages in private industry, there has been a shortage of empirical studies backing the design decisions in VPLs (Whitley, 1997). We will review here the studies that provide evidence for the cognitive processes involved in program comprehension in VPLs.

As the main visual tool available to programmers has been flowcharts, previous studies often focused on whether flowcharts increase comprehension, compared with standard program text. Studies about flowcharts in programming started in the early 1980s and the controversy is still open. Several of them have supported the prediction about the advantages of having flowchart representations (Brooke & Duncan, 1980; Scalan, 1989; Vessey & Weber, 1984) and others have not (Shneiderman, Mayer, MsKay & Heller, 1977; Curtis, Sheppard, Kruesi-Bailey, Baley & Boehm-Davies, 1989).

The results found in these studies showed that the effects of flowchart depended on several variables, mostly related with the characteristics of the task for which the flowcharts were used (Schmalhofer & Glavanow, 1986; Détienne, 1996). In general, flowcharts seemed to help in programming tasks, but only when their use was appropriate. For instance, flowcharts had a more clear effect for those tasks in which programmers needed to follow the control flow, like debugging (Brooke & Duncan, 1980; Curtis *et al*., 1989). Unfortunately, there are no data about the effect of flowcharts in representing other types of information (e.g. data flow information). Flowcharts also seemed to be more effective when they were used alone (Scalan, 1989), than when they were presented as part of the documentation of the program (Schneiderman *et al*., 1977), and for more difficult problems. These results would be coherent with other findings in the literature of visual aids in text learning that have shown that when information depicted pictorially is redundant to the information in the text, or the text is too easy, visual aids do not help (Stein, Brock, Ballard & Vye, 1987).

Green and Petre (1992) and Petre (1995) have conducted a significant work with non-flowchart languages mainly with data flow VPLs. The authors argued that the large part of VPLs effects should be due, not to any intrinsically visual characteristics, but to the "secondary notation" (i.e. aspects of the program layout). More recently, Green and Petre (1996) have applied the cognitive dimensions (Green, 1989) framework to evaluate the effectiveness of VPL and found that the real advantage of VPLs could be that the composition of "plan"-structures is probably easier than in textual languages for two main reasons. First, there are fewer planning goals to be met, such as initialization of variables. Second, the order of activity is freer, so programmers can proceed as seems best. In a recent paper Petre and Blackwell (1999) tried to gather some evidence regarding the effectiveness and usage of VPLs during software design. Using interviews and questionnaires they provided a glimpse of what experts do when they use VPLs for design. They also discussed several techniques for studying mental images in an experimental context. Nevertheless, they did not present any study or further specification about how the proposed techniques could be applied to investigate software design or VPLs.

Spreadsheets are one of the two most important personal computer applications, along with word-processing (Panko, 1988). Among managers, they become the most important (Igbaria, 1992). In short, they are used by millions of users. This massive use is, in part, explained because the HCI community has often lauded spreadsheets, without knowing what the specific advantages are and why. During the first 10 years of the creation of the spreadsheets, several authors emphasized only the strengths of spreadsheets and ignored their weaknesses. These works in general have given some evidence of the advantage of spreadsheets when compared with the traditional imperative programming languages (i.e. Pascal, C) in the effort to complete a simple task (data must be entered, computations performed and/or results displayed). For instance, Norman (1986) claimed that subjects could see better the implication of changing the numbers while for Nardi and Miller (1990) the tabular format of spreadsheets offers great flexibility for representing problems. Unfortunately, most of these authors only gave a speculative analysis or based their views on observational data.

However, we can find some empirical studies that have tried to explain how specific characteristics of spreadsheets are reflected in the programmers' mental representation. Some data, such as those found by Green and Navarro (1995), indicated differences in the programmer's mental representations depending upon the language that they use: BASIC, LabView and spreadsheets.

The most important work in this area has been conducted by Saariluoma and Sajaniemi (1989, 1991, 1994), who addressed the interaction between task and languages using a rigorous methodology, and gave precise explanations for the role of imagery in spreadsheet programming. Because of the length limitation of this paper we will only highlight the main results of this work.

In their first experiment Saariluoma and Sajaniemi (1989) found that several aspects of the superficial organization of the spreadsheet (such as how the reference cells are spread over the sheet or the vertical/horizontal distribution of the cells) caused differences in memory load. The possible explanation proposed was that well-organized layout facilitates visualization of the information and the relationships, helping the user to store this information in chunks. To test this explanation, they investigated in detail, users' strategies for exploring the spreadsheet and concluded that the programmers' search for information was controlled by the surface structure (Saariluoma & Sajaniemi, 1991). Therefore, they suggested that we should discard the rigid-schema directed models for explaining comprehension in VPLs, as proposed for the imperative languages.

In a second study (Saariluoma & Sajaniemi, 1994) they investigated the cognitive processes underlying the transformation of verbal descriptions into mathematical formulas in detail. In four experiments, they found that when spreadsheet users made this transformation they used an intermediate representation. To explain their data Saariluoma and Sajaniemi proposed a theory with multiple processing stages for the transformation of verbal problem instructions into spreadsheet formulas. In the first stage, of the translation process, subjects would analyse the verbal instructions and transform them into a set of propositional representations of instructions for completing the task. After interpreting at least part of the verbal task instruction, subjects try to construct a mental image of the total area. At this stage, one of the functions proposed for images in spreadsheet is to aid the simultaneous verification of the propositional information. In a third stage, subjects would transform the imagery representations into a propositional format and a mathematical representation consisting of a set of operands joined by suitable operators, i.e. formulas. This proposed formula construction process does not need to advance from one stage to another in a strictly serial manner. This theory points to the main role that imagery processing plays in VPL comprehension.

In general, we could draw two conclusions from these empirical results. First, the advantages of VPLs depend upon the task. More specifically, VPLs are better for those tasks in which the integration and use of semantic information is needed. Second, the main cognitive processes underlying the effects of VPLs is Imagery processing. Since there is a large body of research in Cognitive Psychology about the effect of imagery in non-programming tasks, we believe that applying both the results and methodology of this research could allow us to make advances on our understanding of VPL comprehension.

3. Imagery effects

In this section, we will review the main results (and theoretical explanations of them) found in Psychology about how images are processed and used in various tasks. The purpose of this review is to find possible explanations for the effect of imagery on programming with Visual Programming Languages. Interestingly, we will see how the explanation proposed in diverse areas seem to converge in the same direction, pointing out the effect of images in accessing semantic information.

3.1. IMAGE PROCESSING

The research about the processing of pictorial and verbal materials has shown the superiority of pictorial material in memory and comprehension tasks (Paivio, 1977; Potter & Faulconer, 1975). Several theoretical explanations have been proposed for these data, although according to Glaser (1992) they could be classified into two general groups. First, there is a group of theories that emphasize the different structures in which pictures and words could be stored. The more representative theory of this group is the dual coding model of Paivio (1977, 1991), which postulates that the asymmetry in transference of information between systems would cause the superiority of pictorial material in recall, because it would have greater probability of being stored in both systems (redundancy of codes). The second group of theories postulates that there is an amodal and abstract storage to access both pictures and words. To explain the differences between pictures and words within these theories it has been proposed that: (1) there are differences in the access order that both materials have to different information (Potter & Faulconer, 1975; Bajo, 1988); (2) images have a greater distinctiveness and elaboration of visual processing (Nelson, Reed & McEvoy, 1977; Snodgrass & McCullough, 1986); and (3) images have more distinctive semantic representations (Snodgrass & McCullough, 1986).

The debate about the existence of one or two semantic storage systems for images and words is still open (Paivio, 1991). However, since the mid-1980s there have been a lot of results which support an important difference in the access order of verbal and pictorial material to several types of information. Among the wide range of tasks that have been used in these studies, it has been found that response time is quicker for image stimuli than for words. Using a priming task, Bajo (1988) found that while pictures always provide access to the semantic code and produce facilitation, words only provide access to meaning if the instructions emphasize the use of semantic information. Other studies with tasks like Stroop tasks (Glaser & Glaser, 1989) and recall tasks (Nelson *et al.*, 1977) have shown the same pattern of access to semantic information by pictorial material.

As a conclusion to all these studies, without adopting a position about the structures involved, we could affirm that there is different processing for images and words. Pictorial material provides quicker access to semantic information, while verbal material provides quicker access to phonetic and phonologic knowledge. This difference in information access could be the explanation for the effects of pictorial material found in more complex cognitive tasks.

3.2. PICTORIAL AIDS IN TEXT LEARNING

The research about the effectiveness of pictorial aids in text learning has been plentiful. Early work in this area focused on showing the superiority of the conditions with pictorial aids in comparison with the conditions without them. A meta-analysis by Levie and Lentz (1981), which reviewed most of the earlier work, seemed to support the idea that the presentation of pictorial aids facilitated the information that they represented. When the specific effects of different pictorial aids have been studied (MacDonald-Ross, 1978; Alexandrini, 1984), it seems that the effectiveness of pictorial aids depends upon the type of information that they represent.

Based on this conclusion, further studies (Stein *et al.*, 1987; Vicente, 1992) made it clear that the main factor is not the type of relationships depicted in the image, but the congruence between the information depicted and the information that we want to facilitate in the text. For example, Mayer and Gallini (1990) studied the interaction between the information depicted in the image, the information tested and the type of test. His data indicated that there should be congruence among the information depicted in the pictorial aid, the type of information relevant to a specific situation, and the type of test. Thus, if the text is explicative, the graphic aid depicting this information enhances learning, and the type of the test and the information measured should be sensitive to the information represented both in the picture and in the text. These results have been replicated in further studies (Mayer, 1994).

Several explanations have been proposed for the effect of pictorial information in text learning (Stein *et al.*, 1987; Mayer & Gallini, 1990; Hegarty & Just, 1993). For instance, the meta-analysis of 50 experiments of Levie, Anglin and Carney (1987) concluded that pictorial aids enhance text comprehension because they show the critical information needed to organize the information from the text in a meaningful way for the subject. Other authors have claimed that these aids relate the elements in the text, and establish connections between the picture and the diverse information from the text (Mayer, 1989).

Research in this area seems to indicate that pictorial aids enhance the access, integration and comprehension of semantic information. These results are congruent with the findings from image processing research indicating that the information depicted in visual format is codified more efficiently and can enhance learning when we need this information. This conclusion is supported by the data that show the necessity of congruence between the information depicted in the picture and the relevant information in a concrete situation.

3.3. THE IMAGE IN HUMAN–COMPUTER INTERACTION

Research in the area of human–computer interaction (HCI) has mainly focused on the study of user's mental representations from the task or the computer. The general assumption underlying this area of research is that the execution of computer tasks would be enhanced by a good understanding and a good mental model of the system the person is interacting with (Moran, 1981; Norman, 1983). It is assumed that pictorial information enhances the process of mental model development and computer learning. Based on this assumption, numerous interfaces, computer applications, programming environments and programming languages have been created.

Kieras and Bovair (1984) and Kieras (1992) have investigated how and why the interaction between users, usually learners, and the computer could be facilitated by a pictorial interface. In one of their experiments (Kieras & Bovair, 1984), they constructed a model, which showed the functioning of a physical system and the relationships between the system components. The data showed that the group with the model learned and recalled the procedure quicker than the group that learned from a written text. In a second experiment (Kieras & Bovair, 1984), they found that the superiority of the model group was due to the ability of this group to make inferences about the way in which the system works. In further work, Kieras (1992) showed that the ability to make inferences was due to the topological information (i.e. description of the energy flow in the system).

Icons are the more common types of graphic interface in todays computer programs. The common characteristic of all icons is that they are pictorial symbols of an operation, without details that clarify this operation (Rogers, 1989). In general, icons allow a subject to make quicker decisions among several alternatives (Wandmacher & Müller, 1987; Spence & Parr, 1991) in comparison with the verbal commands. A variable that has shown to be important in icon design is the "articulatory distance", the level in which the icon is physically close to the action that it represents, to its semantic meaning. Blankenberger and Hahn (1991) showed that when the location of icons was random, and changed from one trial to another, the execution time of the task increased linearly as the articulatory distance increased. Similar effects of icon location and organization in the execution of computer tasks have been also found in several studies (e.g. Halgren & Cooke, 1993). Therefore, icons seem to be helpful because they facilitate the access to semantic information.

3.4. COMPUTER-ASSISTED INSTRUCTION

In contrast with the abundant research about the use of pictorial aids in traditional instruction methods, there is little work which has empirically investigated the efficiency of these aids in computer assisted instruction (CAI) (Alexandrini, 1987). Most of the work in this area has studied animations (i.e. dynamic images), in contrast to the research reviewed until this point, which has dealt with static images. Both the goals and the conclusions from much of this research are unclear, and difficult to generalize to animations used in other studies. Rieber (1990*a*) reviewed 12 studies with animations with the goal of clarifying the reason for their efficiency. The results of the review seemed to indicate three requirements for animations to be effective. First, animations should be used only when their attributes are congruent with the learning task. That is to say, when the three characteristics of animations (visualization, movement and the path shown by the animated object) are useful in a concrete situation (Rieber, 1990*b*; Park & Gittelman, 1992). Second, learners should perceive the important details shown in the animation (Reed 1985; Rieber, 1989). Finally, another factor which contributes to the animations' efficiency is their interactivity (diSessa, 1982; White, 1984). Navarro-Prieto (1996), in her review of computer animations effects, added another requirement. The type of information measured in the test is important. Animations have shown a clearer superiority when the students were asked for information learned incidentally.

In spite of this effort to clarify the role of animations, we still do not have a theoretical explanation, which explains in which occasions and why these aids enhance learning. However, as in the case of the pictorial aids in text learning, it seems to emerge the necessity of congruence between the information presented and characteristics of the learning situation (e.g. congruency between the information that we want to be learned and the measure that we use to test their effect).

Recent research has started to address this topic. In an empirical study, Navarro, Cañas and Bajo (1996) designed several tutor systems for the MS-DOS operating system, in which several types of pictorial aids and/or text were presented to the subjects. In two experiments, they manipulated the type of information relevant for diverse material. Both the subjects' execution and mental model was measured. Overall, the results from these experiments supported the predictions that pictorial aids enhance learning (in both measures subjects performance and mental representations), but only when they represent the critical information required for a command.

4. Summary

After reviewing all the data about the role of imagery in various areas such as text learning and HCI, and in deeper detail in programming, we can extract two main conclusions. The first is that not all graphical notations are always effective, which leads us to the question of when are they effective. The second makes reference to the explanation of the effects found, in other words what are the differences in processing and why, in VPLs compared with textual languages.

There are several variables that seem to affect the success of visually represented information in programming. One of these variables is the programming task. Overall, flowcharts seem to have a clearer effect for those tasks which require the programmer to trace control flow, like debugging (Curtis *et al*., 1989). The material used in the task has shown to be another important variable. Program visualization techniques seem to be more effective for new problems in which the programmer has no previous training (Shih & Alessi, 1994). These data agree with the research for visual aids both in text learning (Mayer & Gallini, 1990; Mayer, 1994) and in HCI (Navarro-Prieto, 1996), that have shown the importance of the congruence between the information depicted in the picture and the key information of the material for learning. As a consequence, a third variable would be the notation, *per se*, as is postulated for the match–mismatch hypothesis. This hypothesis is based on the principle that the benefits of a notation are relative to a particular task, and every notation highlights some kind of information while obscuring others. Therefore, across the spectrum of information structures, performance would be at its best when the structure of information sought matched the structure of the notation, and that a mismatch would lead to a poor performance. However, we still need a more detailed definition of both notation and information structures. In general, we do not know enough to be able to predict when a concrete visual notation will enhance a programming task.

This introduces our second main question. In order to make sense of all these data and be able to make predictions, we need to explain why visual information is more effective compared with a textual form. Some authors have been trying to answer this question. Petre (1995) proposed that VPLs are effective, not due to an intrinsically pictorial characteristic, but instead due to an effect of the secondary notation, although this secondary notation effect is not clearly defined, and we still have no conclusive data or explanations about its role. Another effort to understand VPL effects, is the cognitive dimensions framework (Green & Petre, 1996) which tries to define the effect of VPLs in the cognitive dimensions framework. Further research should clarify the role of each of these dimensions. We suggest that research about cognitive processes underlying the programming task is urgently needed. An exceptional example of research in this line is the work of Saaruluoma and Sajaniemi (1989, 1994) about spreadsheets. Their data give evidence of the intermediate role of imagery and about the stages and cognitive processes involved in each stage for a very specific sub-task in programming. We claim that to explain the role of imagery in programming, and concretely in VPLs, we need systematic research in the cognitive processes involved in programming.

5. Hypothesis

When we link the two lines of research reviewed up until this point, image processing and psychology of programming, it is possible to propose a hypothesis about the advantages of visual languages over textual languages. On one hand, according to the more accepted theories of program comprehension (Pennington, 1987*b*) programmers go through two phases when they are trying to understand a program. In the first phase, programmers develop a knowledge structure representation, "program model", based on the control flow relationships. In later stages of program comprehension, under appropriate task conditions, programmers develop a plan knowledge representation based on the data flow of the program. This representation contains the main functions of the program, the domain model, and the key information to understand what the program does. It also includes information about the programming situation. The programmers' mental representation seems to depend on his or her experience, the task goal, the length of the program and the programming language characteristics.

On the other hand, basic research in cognitive psychology indicates that there are differences in the processing of pictorial and verbal material. Pictorial material should facilitate quicker access to semantic information, while the verbal material should facilitate quicker access to phonetic information. This is congruent with the data from visual aids in text (Mayer & Gallini, 1990; Mayer, 1994) and HCI learning (Navarro-Prieto, 1996). It has been shown that visual aids enhance learning only when they make access to meaningful information for the situation more likely. Therefore, visual aids are effective because they allow quicker access to the key information for the task. This explanation would also clarify the effects found in visual programming, like the effects of materials and notations in several tasks (e.g. Raymond, 1991).

Therefore, our hypothesis would be that IF the role of imagery is to enhance access to meaningful information THEN VPLs should allow quicker access to data flow information. Therefore, visual programmers should more quickly develop a representation based on data flow relationships, even in easier tasks, in comparison with other non-visual programming languages.

This hypothesis tries to reconcile these two lines of investigation, which have emphasized *either* the effects of the organization of the programmer's knowledge representation *or* the role played by features of the notation of the task language on the emergence, development and support of particular forms of strategy (Davies, 1991).

To test this hypothesis we designed an experiment in which C and spreadsheet programmers are assessed on their mental representations of programs, under different comprehension conditions. Our hypothesis predicts differences in the type of information represented in the programmers' mental model; differences which depend upon the programming language in which the program to be understood is written. Because of that, when thinking about how to test our predictions we reviewed the methodology used to measure mental models in the research areas upon which our hypothesis is based. There have been two research paradigms widely used to study mental representations, which also correspond with the two areas of our theoretical review: (a) the psychology of programming research that has studied programmers' mental representations, and (2) the research in basic psychology investigating image processing. In the psychology of programming knowledge elicitation techniques have been mainly used, while in basic psychology priming paradigms have been predominantly used. Since we were interested in bringing together not only the results but also the methodology of these two areas we will use in our experiment both of these two methodological approaches. We claim that using these methodologies together could give us complementary information that will allow a deeper understanding of both the mental representations and the processes underlying them.

6. Experiment

6.1. METHOD

6.1.1. Participants. Sixty-four programmers with different experience levels participated in the experiment. Thirty-two subjects were C programmers and 32 were spreadsheet programmers.

6.1.2. Apparatus and materials. Four problems were selected to be used in the experiment. Two were typical spreadsheet problems, which calculated the drug consumed for several diseases in a year, and the total carpet and wallpaper cost for a set of rooms. The other two problems were typical C programs, which calculated the first month passing a temperature threshold, and sorted a set of numbers.

Two professional programmers made functionally equivalent versions of the four programs in C and spreadsheet (in Microsoft Excel). The average length of the C programs was 35.5 lines. The spreadsheets had on average 16 rows and 9.25 columns. The two versions of a problem calculated the same thing using the same algorithm, although the implementation was different, following the characteristics of each language. The names of the variables and operations were largely the same for both versions.

For each program a modification task was designed. The criterion to select a modification was that it had to be done in a critical part of the program calculation. Thus, the programmers had to understand what the program does and how to make this modification.

Expert programmers, different from program's authors, were asked to analyse the programs and develop two theoretical nets for each program. One of these theoretical nets contained all the control flow relationships inside this program (control flow theoretical network). The second net that the experts developed for each program specified all the data flow relationships inside this program (data flow theoretical network). These theoretical control flow and data flow networks were our criteria for measuring a subject's level of comprehension of control/data flow information in the program. Examples of these theoretical representations can be found in Appendices A and B.

The experimental tasks were presented on the screen using the general experimental environment (GEE, Sajanieme & Niiranen, 1996). GEE allow us to control the time of the exposition of each stimulus on the screen, and record the response time. For each condition of the experiment a program was built that showed the experimental program, presented the stimulus for the priming and grouping tasks and recorded subject' responses.

6.1.3. Experimental design. For the grouping task, a $2 \times 2 \times 2$ factorial design (language $\times$ control vs. data flow network $\times$ comprehension task) was used, where control vs. data flow network and comprehension tasks (read vs. modify) were within-subject variables and language (C vs. spreadsheet) was a between-subjects variable.

For the priming task a factorial design, $2 \times 4 \times 2$ (comprehension task $\times$ prime condition $\times$ language) was used. Prime condition was a within-subject variable with four levels: data flow related condition, control flow related condition, unrelated condition, non-program condition. Comprehension task (read vs. modify) was also a within-subject factor and language (C vs. spreadsheet) was manipulated between-subjects.

6.1.4. Procedure. First, programmers filled in a questionnaire about their programming experience. Then, they were asked to read two programs (the "read task") and modify two other programs (the "modify task"). The instructions emphasized that they had to read the program until they thought they could understand it (read task) or could make the required modification (modify task). The time limit for both types of comprehension tasks was 10 min. After reading or modifying a program, they performed two tasks designed to elicit the mental representation of the program's structure that they had acquired. First, they performed a primed recognition task and then a grouping task.

6.1.4.1. Primed recognition task. This task has been widely used in image processing research. The motivation for using this task is that it has been shown to be effective in testing whether the subject's mental representations are based on a hypothesized relationship. In each trial, subjects were presented with a program segment (target) taken either from the program that they had read or modified, or from a different program. Their task was to decide as quickly as possible whether or not the segment was part of the program they had already seen. The target segment was preceded by another program segment (prime). The underlying assumption is that knowledge is organized in networks where the activation from one node spreads to nearby nodes. Therefore, if the prime and the target are related in the mental network, the activation of the prime would facilitate the activation of the target. The critical manipulation is the prime–target relationship. To test if the mental model developed by the subjects were based on data or control flow relationships, segments were selected from the two theoretical networks (one control flow network and another data flow network) to create four priming conditions.

1. *Data flow related condition.* A target segment in the test was preceded by a prime close in the theoretical data flow network, and far in the control flow theoretical network.
2. *Control flow related condition.* A target segment in the test was preceded by a prime close in the theoretical control flow network, and far in the data flow theoretical network.
3. *Unrelated condition.* The target segment was preceded by a segment from the same program, but hypothesized to be far away in both the control and data flow theoretical networks.
4. *Non-program condition.* The target segment was from a different program than the prime segment.

The unrelated and non-program were our control conditions because in them all the targets from the related conditions were presented. The unrelated condition allowed us to control that the effects we found were not due to an unspecific activation of the program's representations caused by presentation of a prime from the program. The non-program condition was a base-line for the time needed to answer to a target when the prime does not activate the program's representation (at least not directly).

Half of the targets were part of the original program. The other half were not from the program, therefore the subject should reject them as not belonging to the original program. These non-program targets were constructed by modifying some program segments, so that the information in the modified fragment did not correspond with the information shown in the original program. In order to be sure that the data flow related condition was not affected by control flow information (and the other way around), two controls were done with the material. First, in the data flow conditions, we replaced the operations in these segments with dots. Following the same reasoning in the control flow conditions, information about the name of the variables was replaced with dots. Second, a distracter code fragment was presented with every target. The target and the distracter were identical except in an operation, in the case of the control flow condition, or a variable name in case of the data flow condition or one of these options for the rest of conditions. The role of the distracter is to ensure that the recognition decision had to be based on the exact recognition of the data or control flow information from the program and not on that this fragment looks familiar. Also, the spatial location of the target and distracter segments on the screen was balanced. Thus, the subjects could not associate a spatial location on the screen with the stimulus they should select.

Primes were presented for 10 s, so that participants had time to read them. During this time, no answer was required from the participant, who was instructed to carefully read the prime segments. This time was calculated in a pilot study as the maximum time needed to read the segments. After the prime disappeared, the next screen presented the target and distracter segments until the subject responded. In this screen, subjects were asked to click on the segments that they thought were from the original program, or if neither of them was from the original program, click the OK button to go to the next trial.

Recognition accuracy and time were recorded. We predicted a priming effect. Response times to the target segment preceded by a prime close in the network structure should be faster than response time (and with better accuracy) to the same target preceded by a prime which was not as close in the cognitive structure. This priming effect would be observed in the control and/or data flow conditions depending on the knowledge acquired by the subject.

6.1.4.2. Grouping task. This task has been used successfully to access a subjects' mental model in programming (e.g. Robertson & Yu, 1990). The participants were presented with a program that they had read or modified previously, and were asked to group together the lines of code or the cells that they thought were related to each other. To make a group they just needed to click on the lines/cells that they wanted to select. They could make as many groups as they found important. They were given some practice trials for this task with names of fruits and mountains.

6.2. RESULTS

Two expert programmers quantified the data from the experience questionnaire. We wanted the experience data from each subject to be sure that our results were not interacting with this variable, which has been shown to be important in developing mental representations in programming (Corritore & Wiedenbeck, 1991). The significance level adopted for all the analysis is 0.05.

6.2.1. Grouping task results. Raw grouping data were transformed into proximity data by calculating the number of times that two segments of the program were grouped together by one subject. Therefore, for each subject there were four proximity matrices, one for each program (two belonging to the modify condition and two belonging to the read condition).

Those matrices were submitted to a PathFinder analysis (Schvaneveldt, 1990). Pathfinder is a graph-theoretic technique that derives network structures from proximity data. Pathfinder algorithm takes proximity matrices and produces a network in which concepts are represented as nodes and relations between concepts are represented as links between nodes.

The PathFinder analysis provided us with a measure of the similarities among networks called *C*† (ranging from 0 to 1). We calculated the *C* between each subject's network for each program and our theoretic control (four *Cs*) and data flow networks (four *Cs*) that were our criteria for measuring a subject's level of comprehension of control/data flow information in the program.

In total, four *C* values were calculated for each participant, two for the modification task and two for the reading task. Then, *Cs* were averaged in each task so that we ended up with two *Cs* for each participant, one for the modification task and one for the reading task.

†We are going to underly this C measure to differentiate it from the programming language with the same name.

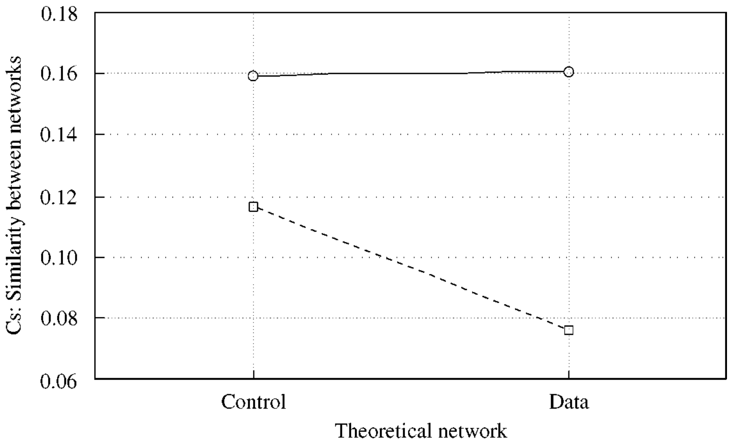


FIGURE 1. Grouping results: closeness (C parameter) of the C and spreadsheet programmer networks to the control and data flow theoretical networks: —○—, Language spreadsheet; – – □ – –, Language C.

A factorial ANOVA, 2 × 2 × 2 (language × control vs. data flow network × comprehension task) was performed. Language was a between-subjects variable and control vs. data flow network, and comprehension tasks were within-subject variables. The level of experience of the participants was a covariant variable in this analysis.

Results showed significant effects of the programming language, the priming condition and the interaction between them. With regard to the effect of programming language [$F(1,60) = 108.934$, $MSE = 0.0023$, $p = 0.000$], spreadsheet programmer representations were closer to both the control and data flow criteria ($C = 0.160$) than C programmer's mental representation ($C = 0.096$).

On the other hand, these two types of information were learned differently among programmers [$F(1,61) = 35.1022$, $MSE = 0.0006$, $p = 0.000$]. Overall the programmers' mental representations were closer to the control flow theoretical networks ($C = 0.137$) than the data flow theoretical network ($C = 0.118$).

Especially relevant to our hypothesis was the interaction between the programming language and the theoretical network [$F(1,61) = 41.7506$, $MSE = 0.0006$, $p = 0.0000$, $LSD = 0.013$], as can be seen in Figure 1. As we can see in Figure 1, C programmers had better networks for the Control Flow information compared with the Data Flow information. In accordance with our predictions, spreadsheet programmers seem to have developed better mental structures for both Control and Data Flow information as compared to C programmers.

6.2.2. Discussion. According to our hypothesis, C programmers have better networks for control flow information compared with data flow information. So, C programmers learned the control flow information better than data flow information. On the other hand, spreadsheet programmers seem to have developed good mental structures for both control and data flow information.

6.2.3. Primed recognition task results. The number and average time of correct responses were recorded for positive and negative trials (where the target segment was part or not

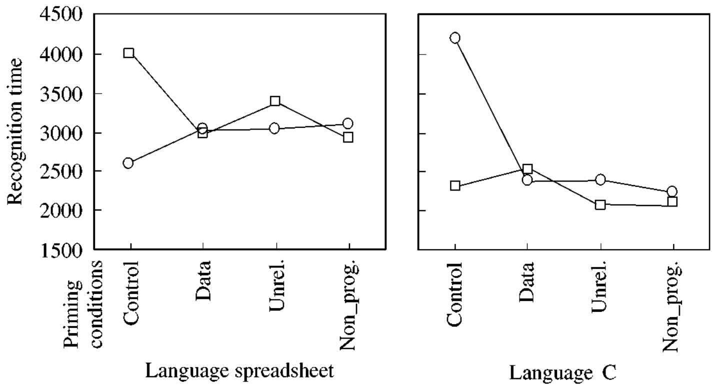


FIGURE 2. Priming recognition results: response times to positive trials, second order interaction: programming language by comprehension task and by prime condition: —○—, Task modify; —□—, Task read.

part of the studied program, respectively), separately for each participant. An ANOVA $2 \times 4 \times 2$ (comprehension task $\times$ prime condition $\times$ language) was used. Language was a between-subjects variable, and comprehension task and prime condition were within-subject variables. The level of experience of the participants was again a covariant variable.

6.2.3.1. Positive Trials. (in which the target segment was from the studied or modified program)

Recognition time. The effect of Language was significant [$F(1,61) = 7.332$; $p = 0.008$]. C programmers were faster (average = 2542 ms) than spreadsheet programmers (average = 3137 ms). Although the interaction between the language and the comprehension task was not significant [$F(1,61) = 3.49$, $p = 0.067$], it looked promising for further research. It is interesting to notice that in that interaction C programmers showed larger differences between tasks than spreadsheet programmers.

This interaction was modulated by the effect of the significant second-order interaction of prime condition by comprehension task and language [$F(3,186) = 3.30$; $p < 0.021$, $LSD = 1265$]. As can be seen in Figure 2, when C programmers modified a program, control primes slowed down recognition times compared with data, program unrelated and non-program. There were no differences among primes when C programmers had to read a program. For spreadsheet programmers the data followed the opposite pattern, although the differences were not significant. Control primes tended to slow down recognition after reading, and made recognition faster after modifying a program faster.

Accuracy. There were no significant effects of any of the manipulated variables. The lack of significant effects indicated that recognition times results were not due to a trade-off between speed and accuracy in the responds. The percentage of errors was around 30%.

6.2.3.2. Negative Trials. (in which the target segment was NOT from the studied or modified program)

Recognition time. The effect of language was significant [$F(1,61) = 14.6$; $p = 0.0003$]. Again, C programmers were faster (average = 2135 ms) than spreadsheet programmers

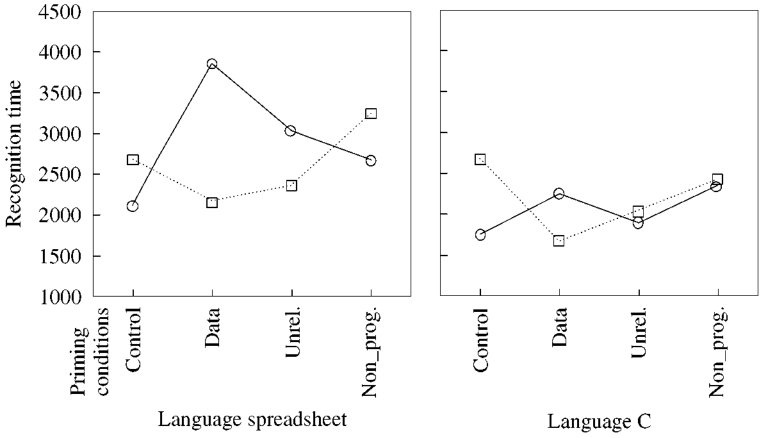


FIGURE 3. Priming recognition results: response times to negative trials, second-order interaction: programming language by comprehension task and by prime conditions task: —○—, modify; ···□···, read.

(average = 2768 ms). The interaction of prime condition by comprehension task was also significant [$F(3,186) = 10.72$; $p < 0.001$]. The data showed that when programmers read a program, the control flow condition slowed down the recognition process in comparison with the other conditions. In the opposite way, when programmers had done a modification task the data flow targets needed more time for recognition. It seems that programmers have acquired the control flow information in the reading task, and activating this information in the control flow condition made it more difficult to refuse the incorrect target. The same interference effect seemed to happen with the data flow information for the modification task.

The effect of this interaction seemed to be modulated by a second-order interaction (Figure 3), of language by prime condition and comprehension task. Again, although this interaction does not reach the significance level [$F(3,186) = 1.97$, $p < 0.1204$, $LSD = 671$], we will discuss this interaction because of its importance for further research.

When C programmers had to read the program there was an inhibition effect of the control flow conditions in comparison with the data flow conditions. On the other hand, when C programmers modified the programs, there was a tendency for the data flow conditions to increase the time needed for recognition although this difference was not significant. With regard to the spreadsheet programmers, they also performed differently depending on the task. There were no differences among prime conditions when they had to read a program. On the contrary, after modifying the programs, data flow and control flow conditions had the opposite effects. Data flow conditions slowed down the process of rejecting a fragment, in comparison with the control, program unrelated and no program conditions. Control flow conditions seemed to need less time to take the correct decision in comparison with the data flow and program unrelated conditions.

Accuracy. There was a significant effect of prime condition [$F(3,186) = 5.20$; $p < 0.002$]. Target segments that belonged to the studied program (data and control flow conditions and program unrelated condition) were recognized better than targets that belonged to different programs.

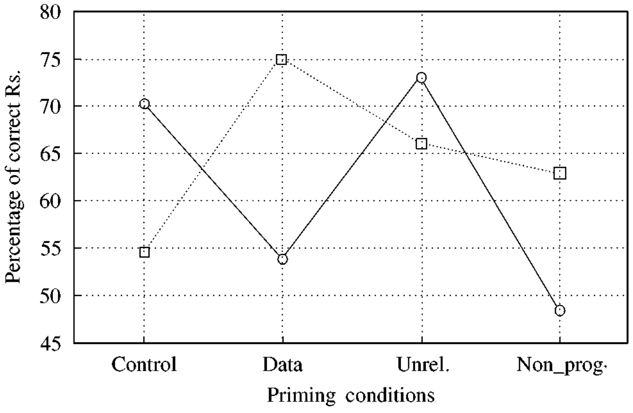


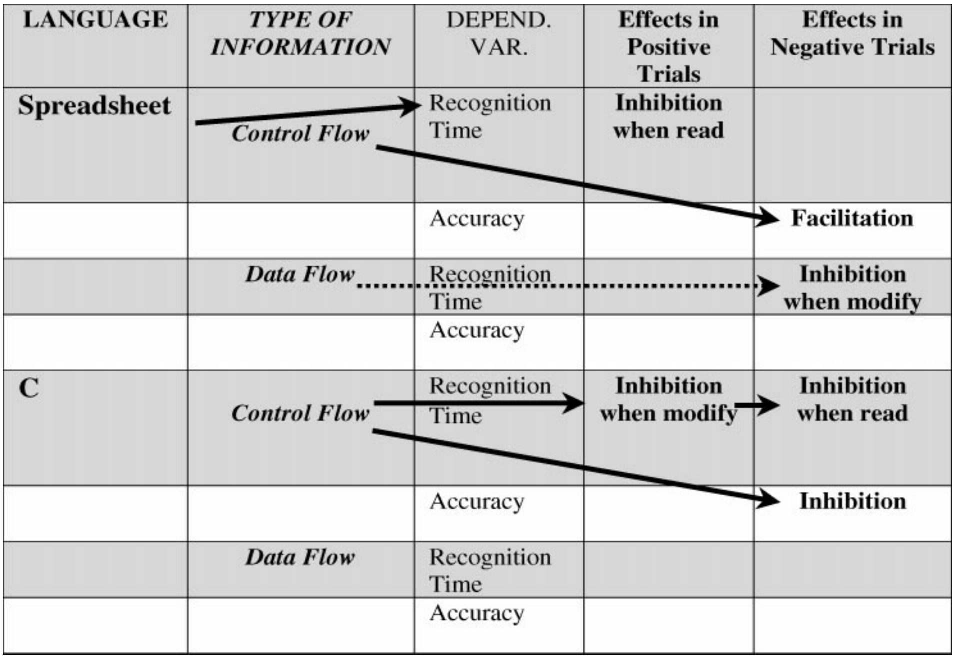
FIGURE 4. Priming recognition results: percentage of correct responses to negative trials, second-order interaction: programming language by prime condition. Language: —○—, Spreadsheet; ···□···, C.

The interaction of language by prime condition was also significant [$F(3,186) = 11.91$; $p < 0.000$; $MSE = 811.5$, $LSD = 10$]. As we can see in Figure 4, for C programmers control flow conditions decreased accuracy compared to the data flow and non-program conditions. On the other hand, for spreadsheet programmers data flow conditions decreased accuracy compared to control flow or no program conditions. So, control flow conditions inhibited recognition for C programmers and data flow conditions inhibited recognition for spreadsheet programmers.

6.2.4. Discussion. From all these data, we could conclude that, subjects showed an inhibitory effect pattern different for each language, especially for the negative trial. Because of the number of variables manipulated, and their complexity, our conclusions will be based on the general pattern found, rather than each of the individual effects. To facilitate the interpretation of our results, we have summarized them in Table 1. The conclusions that can be deduced from our results will be divided by the two types of information that the prime recognition task provided, as we discussed previously in the methodology section.

First, the results in the prime recognition tasks confirm our hypothesis about the influence of the programming language on the activation and acquisition of the data flow and control flow information. In both types of trials, significant effects of the control flow information for C programmers, and both types of information for spreadsheet programmers were found. These data paralleled the data found in the grouping task, indicating that spreadsheet programmers not only have acquired both types of information, but also that they activated both of them in the recognition task. In contrast, C programmers, although they seemed to have acquired some information about the program data flow (as we saw in the data from the grouping task), did not activate this information in either of the two types of trials. This difference between programmers seemed to indicate that C programmers follow a strategy focused on control flow information when they were trying to retrieve the information from the program needed for the recognition task.

Table 1
Summary of the effects found in the primed recognition tasks (empty cells = no effects). The line arrows represent the effects found under the control flow related conditions and the dotted arrows represent the effects found under the data flow related conditions



LANGUAGE	*TYPE OF INFORMATION*	DEPEND. VAR.	Effects in Positive Trials	Effects in Negative Trials
Spreadsheet	*Control Flow*	Recognition Time	**Inhibition when read**	
		Accuracy		**Facilitation**
	Data Flow	Recognition Time		**Inhibition when modify**
		Accuracy		
C	*Control Flow*	Recognition Time	**Inhibition when modify**	**Inhibition when read**
		Accuracy		**Inhibition**
	Data Flow	Recognition Time		
		Accuracy		

Again, these data followed a regular pattern, which indicated that when a knowledge structure was activated, the result was an interference, specifically when the programmers needed to reject a target which was not from the original program (negative trials). This was the case for data flow information for spreadsheet programmers (especially after a modification task) and for control flow information for C programmers. Therefore, we have found that the strategic focus on one type of information was affecting the execution in the recognition task. Concretely, it seemed that focusing on a specific type of information produced interference to recognize a fragment.

All these data seemed to show that we could not interpret the execution of the subjects as a simple spread of activation. To fully understand the data we need to pay attention to what the programmer is really doing during the task with the information that we give to them. In light of our results, the programmer seems to be doing a strategic task to integrate information from the prime and information from the target, to answer recognition questions efficiently. However, C and spreadsheet programmers seemed to base their strategies on different information. While C programmers focused on Control Flow information, spreadsheet programmers focused on Data Flow information. This strategic process would be more complex when the amount of information the programmer must pay attention to increases. Thus, an increase in task complexity would explain the interference data found in the experiment.

7. General discussion

The present research is relevant both from a theoretical and an applied point of view. From a theoretical perspective the data from the experiment support: (1) the role postulated for imagery in program comprehension, and (2) the methodology used.

The conclusions are derived from the two tests that were used to assess a subject's mental representations. In general, the results showed evidence that imagery influences programmers' mental representations. Specifically, there were differences in the information acquired by the programmers depending upon the language and the comprehension task. These results were congruent with previous research showing differences in the program representation depending on the task and programming language (Pennington, 1987*a*; Burkhardt *et al*., 1997). The data from the grouping task showed that the mental structure of the spreadsheet programmers had more information about both control and data flow. However, C programmers seemed to have a better mental representation for the control flow information than they had for the data flow information. With regard to the primed recognition task, there was a strong interaction effect between programming language and prime condition when the programmers had to reject a target that was not from the program. C programmers seemed to be more influenced by the control flow primes, especially after a reading task, while spreadsheet programmers seemed to activate both types of information, but they showed a strong interference under the data flow conditions when they had modified the programs.

Furthermore, these data together gave evidence supporting our hypothesis that the spreadsheet, with its visual characteristics, helped programmers develop a representation of the program based on data flow structure. The grouping data clearly showed that the spreadsheet programmers develop a data flow mental representation even in the easiest tasks, while C programmers seemed to have more problems acquiring and using this information. According to this interpretation, the positive effects found for VPLs could be explained because the images optimize access to semantic information. This role of images could also explain the contradictory data found in this area, because from it we could deduce that images would not always be helpful, rather congruency is necessary between the information shown and the information relevant in a concrete situation.

Our data with C programmers replicated the results of Pennington (1987*a*), supporting her two stages theory for procedural languages. On the other hand, based on the effects found for spreadsheet programmers, we claim that it is necessary to add a factor to this theory, to account for the role of imagery in programming. We suggest that the stages proposed for program comprehension should be more flexible, depending on the variables that could enhance the processes underlying these stages. One of these variables seems to be imagery, which enhances access to data flow information. In order to have a complete theory about program comprehension we need to investigate the underlying cognitive processes in more detail and study how they could be affected by several variables.

From an applied perspective, this research is relevant for educational software design, the design of new programming languages, and how to teach them. In the light of our data we can conclude that the format of the information presented is very important for the process of learning this information. Although we need more research to specify which visual characteristics of the spreadsheet influence the comprehension process; to present the information in a pictorial format seems to enhance the access to relevant program information. There was also a significant effect of the task done with the information to be learnt. With regard to the teaching of programming, we suggest the inclusion of images which represent the relevant program information, and to emphasize that the learners perform tasks which require access to this information. On the other hand, the results, and explanation proposed for the effects of the VPLs, point out that the designers of programming languages need to consider that imagery does not always enhance comprehension. In our previous work, we have discussed the necessity of first evaluating the semantic information relevant to each situation, and depicting only this information in the images (Navarro *et al*., 1996).

Another important contribution of this experiment was to propose a new methodology to measure programmer's mental models. This methodology resulted from the combination of primed recognition and knowledge elicitation techniques. Psychology of programming could gain much from these methodologies which have been used successfully in experimental psychology research for many years. We claim that using these methodologies together could give us complementary information, which allows a deeper understanding of both the mental representations and the processes underlying them. In our experiment, the uses of knowledge elicitation techniques, such as the grouping task, allowed us to build subject representation networks, and test how similar they were with the hypothesized ones. With the information obtained in this task we were able to test whether programmers acquired a type of information or not. On the other hand, the primed recognition task provided detailed time and accuracy data about when these relationships were used to make decisions about the original code. The priming effects found also supported the usefulness of our control and data flow theoretical analysis of the programs, and thus the usefulness of these analyses with the grouping data.

The different information provided by each technique would explain the difference found in the results from each of them. The data indicate that the primed recognition paradigm is more sensitive than the grouping task to the type of task that the programmers performed. These data were explainable because with this paradigm we were measuring access to some relationships. Thus, although programmers had represented both types of information, the access to the most consolidated information would be quicker.

We would like to point out that our data support this combined methodology as a useful way to study complex cognitive tasks. We consider that this methodology is an option which could address the present problem about how to measure mental models in complex tasks.

There were two main limitations of this study that we would like to control in further research. First, it would be important to manipulate the program length, because this variable has been shown to influence program comprehension (Pennington, 1987*a*; Corritore & Wiedenbeck, 1991). In our experiment, we used relatively small programs (with an average of 40 lines). It would be interesting to investigate how our results could be generalized to longer programs. The manipulation of longer programs (closer to the industry size programs) would also give more ecological validity to our data. In the same way, it would be very interesting to study how these results could be generalized to other VPLs.

Second, the next logical step would be to study the effects of the diverse spreadsheet characteristics in detail. Further research should examine if the differences found between C and spreadsheet programmers were actually due to the visual characteristic of spreadsheets, and which of them were more important for the program comprehension process.

References

ALEXANDRINI, K. L. (1984). Pictures and adult learning. *Instructional Science*, **13,** 63–77.

ALEXANDRINI, K. L. (1987). Computer graphics in learning and instruction. In H. A. HOUGHTON & WILLOWS, Eds. *The Psychology of Illustration*, Chapter 6, pp. 160–188. New York: Springer-Verlag.

BAJO, M. T. (1988). Semantic facilitation with pictures and words. *Journal of Experimental Psychology: Learning, Memory and Cognition*, **4,** 579–589.

BLANKENBERGER, S. & HAHN, K. (1991). Effects of icon design on human–computer interaction. *International Journal of Man–Machine Studies*, **5,** 363–377.

BROOKE, J. B. & DUNCAN, K. D. (1980). Experimental studies of flowchart use at different stage of program debugging. *Ergonomics*, **23,** 1057–1091.

BURKHARDT, J. M., DÉTIENNE, F. & WIEDENBECK, S. (1997). Mental representations constructed by experts and novice in object-oriented program comprehension. *INTERACT'97*. Sydney, Australia, July 14–18.

CAÑAS, J. J., BAJO, M. T. & GONZALVO, P. (1994). Mental models and computer programming. *International Journal of Human–Computer Studies*, **40,** 795–811.

CORRITORE, C. L. & WIEDENBECK, S. (1991). What do novices learn during program comprehension? *International Journal of Human–Computer Interaction*, **3,** 199–222.

CORRITORE, C. L. & WIEDENBECK, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human–Computer Studies*, **50,** 61–84.

CUNNIF, N. & TAYLOR, R. P. (1987). Graphical vs. textual representations: an empirical study of novices' program comprehension. In F. M. OLSON, C. SHEPPARD & E. SOLOWAY, Eds. *Empirical Studies of Programmers: Second Workshop*, pp. 114–129. Norwood, NJ: Ablex.

CURTIS, B., SHEPPARD, S. B. KRUESI-BAILEY, E., BALEY, J. & BOEHM-DAVIES, D. A. (1989). Experimental evaluation of software documentation formats. *Journal of Systems and Software*, **9,** 167–207.

DAVIES, S. P. (1991). The role of notation and knowledge representation in the determination of programming strategy: a framework of integrating models of programming behaviour. *Cognitive Science*, **15,** 547–572.

DAVIES, S. P. (1994). Knowledge restructuring and the acquisition of programming expertise. *International Journal of Man–Machine Studies*, **40,** 703–726.

DEGROOT, A. D. (1965). *Thought and Choice in Chess*. Paris: Mounton.

DÉTIENNE, F. (1996). What model(s) for program understanding? *Proceedings of the Conference on Using Complex Information Systems (UCIS'96)*, Poitiers, France, September 4–6.

DISESSA, A. (1982). Unlearning Aristotelian Physics: a study of knowledge-based learning. *Cognitive Science*, **6,** 37–75.

GILMORE, D. J. & GREEN, T. R. G. (1984). Comprehension of miniature programs. *International Journal of Man–Machine Studies*, **21,** 31–48.

GILMORE, D. J. & GREEN, T. R. G. (1988). Programming plans and programming expertise. *The Quarterly Journal of Experimental Psychology*, **40A,** 423–442.

GLASER, E. R. & GLASER, M. O. (1989). Contex effects in stroop-like word and picture processing. *Journal of Experimental Psychology: General*, **118,** 13–42.

GLASER, W. R. (1992). Picture naming. *Cognition*, **42,** 61–105.

GOLIN, E. J. (1991). Theory of visual languages. *Journal of Visual Languages and Computing*, **2,** 309–310.

GREEN, T. R. G. (1989). Cognitive dimensions of notation. In R. WINDER & A. SUTCLIFFE, Eds. *People and Computers V*. Cambridge: Cambridge University Press.

GREEN, T. R. G. & NAVARRO, R. (1995). Programming plans, imagery and visual programming. In K. NORDY, P. H. HELMERSEN, D. J. GILMORE & S. A. ARNESEN, Eds. *INTERACT-95*, pp. 139–144. London: Chapman & Hall.

GREEN, T. R. G. & PETRE, M. (1992). When visual programs are harder to read then textual programs. *Proceedings of the 6th Conference on Cognitive Ergonomics ECCE6*, pp. 167–180.

GREEN, T. R. G. & PETRE, M. (1996). Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, **7,** 131–174.

HALGREN, S. L. & COOKE, N. J. (1993). Towards ecological validity in menu research. *International Journal of Man–Machine Studies*, **39,** 51–70.

HEGARTY, M. & JUST, M. A. (1993). Constructing mental models of machines from text and diagrams. *Journal of Memory and Language*, **32,** 717–742.

HENDRY, D. G. & GREEN, T. R. G. (1994). Creating, comprehending, and explaining spreadsheets: a cognitive interpretation of what discretionary users think of the spreadsheet model. *International Journal Human-Computer Studies*, **40,** 1033–1065.

IGBARIA, M. (1992). An examination of microcomputer use in Taiwan. *Information and Management*, **22,** 19–28.

KAY, A. (1984). Computer software. *Scientific American*, September.

KIERAS, D. (1992). Diagrammatic displays for engineered systems: effects on human performance in interacting with malfunctioning systems. *International Journal of Man–Machine Studies*, **36,** 861–895.

KIERAS, D. E. & BOVAIR, S. (1984). The role of mental model in learning to operate a device. *Cognitive Science*, **8,** 255–273.

KIPPER, J. D., HOWARD, E. & AMES, C. (1997). Criteria for evaluation of visual programming languages. *Journal of Visual Languages and Computing*, **8,** 175–192.

LEVIE, W. H., ANGLIN, G. J. & CARNEY, R. N. (1987). On empirical validating functions of pictures in prose. In H. A. HOUGHTON & D. A. WILLOWS, Eds. *The Psychology of Illustration*, Chapter 2, pp. 51–86. New York: Springer-Verlag.

LEVIE, W. H. & LENTZ, R. (1981). Effects of text illustration: a review of research. *Educational Communication Technology*, **30,** 195–232.

MAYER, R. E. (1976). Some conditions of meaningful learning for computer programming: advance organisers and subject control of frame order. *Journal of Educational Psychology*, **68,** 143–150.

MAYER, R. E. (1980). Elaboration techniques of technical text: an experimental test of the learning strategy hypothesis. *Journal of Educational Psychology*, **72,** 770–784.

MAYER, R. E. (1989). Systematic thinking fostered in illustration in scientific text. *Journal of Experimental Psychology*, **81,** 240–246.

MAYER, R. E. (1994). Visual aids to knowledge construction: building mental representations from picture and words. In W. SCHNOTZ & R. KULHAVY, Eds. *Comprehension of Graphics*, pp. 125–138. Amsterdam: North-Holland.

MAYER, R. E. & BROMAGE, B. (1980). Different recall protocols for technical text due to advance organisers. *Journal of Educational Psychology*, **72,** 209–225.

MAYER, R. E. & GALLINI, J. K. (1990). When is an illustration worth ten thousand words? *Journal of Experimental Psychology*, **4,** 715–726.

MCDONALD-ROSS, M. (1978). Graphics in text. *Review o Research in Education*, **5,** 47–83.

MOHER, T. G., MAK, D. C., BLUMENTHATL, B. & LEVENTHAL, L. M. (1993). Comparing the comprehensibility of textual and graphical programs: the case of Petri nets. In C. R. COOK, J. C. SCHOLTZ & J. C. SPOHRER, Eds. *Empirical Studies of Programmers: Fifth Workshop*, pp. 137–161. Norwood, NJ: Ablex.

MORAN, T. P. (1981). A applied psychology of the user. *Computing Surveys*, **13,** 1–11.

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

NARDI, B. A. & MILLER, J. R. (1990). The spreadsheet interface: a basis for end user programming. *INTERACT*'90, pp. 977–983. Amsterdam: Elsevier.

NARDI, B. A. & MILLER, J. R. (1991). Twinkling lights and nested loops: distributed problem solving and spreadsheets development. *International Journal of Man–Machine Studies*, **34,** 161–184.

NAVARRO-PRIETO, R. (1996). El Efecto de la Ayudas Pictóricas en la Interacción con el Ordenador. Unpublished Master Dissertation.

NAVARRO-PRIETO, R., CAÑAS, J. J. & BAJO, M. T. (1996). Pictorial aids in computer use. In T. R. G. GREEN, J. J CAÑAS & C. WARREN, Eds. *Proceedings of the 8th European Conference on Cognitive Ergonomics*, pp. 77–82, Granada.

NELSON, D. L., REED, V. S. & MCEVOY, C. L. (1977). Learning to order pictures and words: a model of sensory and semantic encoding. *Journal of Experimental Psychology*: *Human Learning and Memory*, **3,** 485–497.

NORMAN, D. A. (1983). Some observations on mental models. In D. GETNER & L. A. STEVENS, Eds. *Mental Models*. Hillsdale, NJ: LEA.

NORMAN, D. A. (1986). Cognitive engineering. In D. A. NORMAN & S. W. DRAPER, Eds. *User-Centred Systems Design*, pp. 31–61. Hillsdale, NJ: Lawrence Erlbaum.

PAIVIO, A. (1977). Images, propositions and knowledge. In J. M. NICHOLAS, Ed. *Images*, *Perception*, *and Knowledge*. The Western Ontario Series in the Philosophy of Science. Dordrecht: Riedel.

PAIVIO, A. (1991). Dual coding theory: retrospect and current status. *Canadian Journal of Psychology*, **45,** 255–287.

PANKO, R. (1988). Object-oriented spreadsheets: the analytic spreadsheet package. *Proceedings of OOPSLA*'*86*, pp. 385–390.

PARK, O. & GITTELMAN, S. S. (1992). Selective use of animation and feedback in computer-based instruction. *Educational Technology Research and Development*, **40,** 1042–1629.

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

PETRE, M. & BLACKWELL, A. F. (1999). Mental imagery in program design and visual programming. *International Journal of Human Computer Studies*, **51,** 7–30.

PENNINGTON, N. (1987*a*). Stimulus structures and mental representation in expert comprehension of computer programs. *Cognitive Psychology*, **19,** 295–341.

PENNINGTON, N. (1987*b*). Comprehension strategies in programming. In F. M. OLSON, C. SHEPPARD & E. SOLOWAY, Eds. *Empirical Studies of Programmers*: *Second Workshop*. Norwood, NJ: Ablex.

POTTER, M. C. & FALCONER, B. A. (1975). Time to understand pictures and words. *Nature*, **253,** 437–438.

RAYMOND, D. (1991). Characterising visual languages. *The 1991 IEEE Workshop on Visual Languages*. Silver Spring, MD: IEEE Computer Society Press.

REED, S. (1985). Effect of computer graphics on improving estimates to algebra word problems. *Journal of Educational Psychology*, **77,** 285–298.

RIEBER, L. P. (1989). Animation in computer-based instruction. *Educational Technology Research and Development*, **38,** 77–86.

RIEBER, L. P. (1990*a*). Using computer animated graphics in science instruction with children. *Journal of Educational Psychology*, **82,** 135–140.

RIEBER, L. P. (1990*b*). The effects of computer animation on adult learning and retrieval tasks. *Journal of Computer-Based Instruction*, **17,** 46–52.

RIST, R. S., (1986). Plans in programming: definition, demonstration and development. In E. SOLOWAY & I. YEGER, Eds. *Empirical Studies of Programmers*, pp. 28–46.

ROBERTSON, S. P., DAVIES, E. F., OKABE, K. & FITZ-RANDOLF, D. (1990). Program comprehension beyond the line. In D. DIAPER *et al*., Eds. *Human–Computer Interaction. INTERACT*'*90*. Amsterdam: Elsevier Science Publishers B. V, North-Holland.

ROBERTSON, S. P. & YU, C. C. (1990). Common cognitive representations of program code across task and languages. *International Journal of Man–Machine Studies*, **33,** 343–360.

ROGERS, Y. (1989). Icons at the interface: their usefulness. *Interacting with Computers*, **1,** 105–117.

RUMELHART, D. E. (1980). A reply to Black and Wilensky. *Cognitive Science*, **4,** 313–316.

RUMELHART, D. E. & NORMAN, D. A. (1978). Accretion, tuning and restructuring: three models of learning. In J. W. COTTON & R. KLATSKY, Eds. *Semantic Factors in Cognition.* NJ: Erlbaum.

SAARILOUMA, P. & SAJANIEMI, J. (1989). Visual information chunking in spreadsheet calculation. *International Journal of Man–Machine Studies*, **30,** 475–488.

SAARILOUMA, P. & SAJANIEMI, J. (1991). Extracting implicit tree structures in spreadsheet calculation. *Ergonomics*, **34,** 1027–1046.

SAARILOUMA, P. & SAJANIEMI, J. (1994). Transforming verbal descriptions into mathematical formulas in spreadsheet calculation. *International Journal of Human–Computer Studies*, **41,** 915–948.

SAJANIEMI, J. & NIIRANEN, T. (1996). *GEE user's manual . Technical Report A-1996-1*, Department of Computer Science, University of Joensuu.

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

SCHVANEVELDT, R. W. (1990) *PathFinder Associative Networks*: *Studies in Knowledge organization.* Norwood. N.J.: Ablex Publishing Corporation.

SCHANK, R. C. & ABELSON, R. P. (1977). *Scripts*, *Plans*, *Goals*, *and Understanding*. Hillsdale, NJ: Lawrence Erlbaum Associates.

SCHMALHOFER, F. & GLAVANOV, D. (1986). Three components of understanding a programmer's manual: verbatim, propositional, and situational representations. *Journal of Memory and Language*, **25,** 279–294.

SHIH, Y-F. & ALESSI, S. M. (1994). Mental model and transfer of learning in computer programming. *Journal of Research in Computing Education*, **26,** 154–175.

SHNEIDERMAN, B. (1976). Exploratory experiments in programmer behaviour. *International Journal of Computer and Information Sciences*, **5,** 124–143.

SHNEIDERMAN, B., MAYER, R., MSKAY, D. & HELLER, P. (1977). Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, **20,** 373–381.

SNODGRASS, J. G. & MCCULLOUGH, B. (1986). The role of visual similarity in picture categorization. *Journal of Experimental Psychology*: *Learning*, *Memory and Cognition*, **12,** 147–154.

SPENCE, R. & PARR, M. (1991). Cognitive assessment of alternatives. *Interacting with Computers*, **3,** 270–282.

SOLOWAY, E. & EHRLICH, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering*, **5,** 595–609.

SOLOWAY, S., EHRLICH, K. & BLACK, J. B. (1983). Beyond numbers: don't ask "how many" as "why". *Proceedings of the Conference of Human Factors in Computer Systems*, Boston, MA.

SOLOWAY, S., EHRLICH, K. & BONAR, J. (1982). Tapping into tacit programming knowledge. *Proceedings of the Conference on Human Factors in Computer Systems*, Gaithersburg, MD.

STEIN, B. S., BROCK, K. F., BALLARD, D. N. & VYE, N. J. (1987). Constraints on effective pictorial and verbal elaboration. *Memory and Cognition*, **15,** 281–290.

VAN DIJK, T. A. & KINTSCH, W. (1983). Tapping into tacit programming knowledge. *IEEE Transactions on Software Engineering*, **SE-10,** 595–609.

VESSEY, I. & WEBER, R. (1984). Research on structured programming: an empiricist's evaluation. *IEEE Transactions of Software Engineering*, **SE-10,** 397–407.

VICENTE, K. J. (1992). Memory recall in a process control system: memory measure of expertise and display effectiveness. *Memory and Cognition*, **20,** 356–373.

WANDMACHER, J. & MÜLLER, U. (1987). On the usability of verbal and iconic command representations. *Zeitschift für Psychologie*, **9** (Suppl), 35–45.

WIEDENBECK, S. & RAMALINGAM, V. (1999). Novice comprehension of small programs written in the procedural and object-oriented styles. *International Journal of Human Computer Studies*, **51,** 71–88.

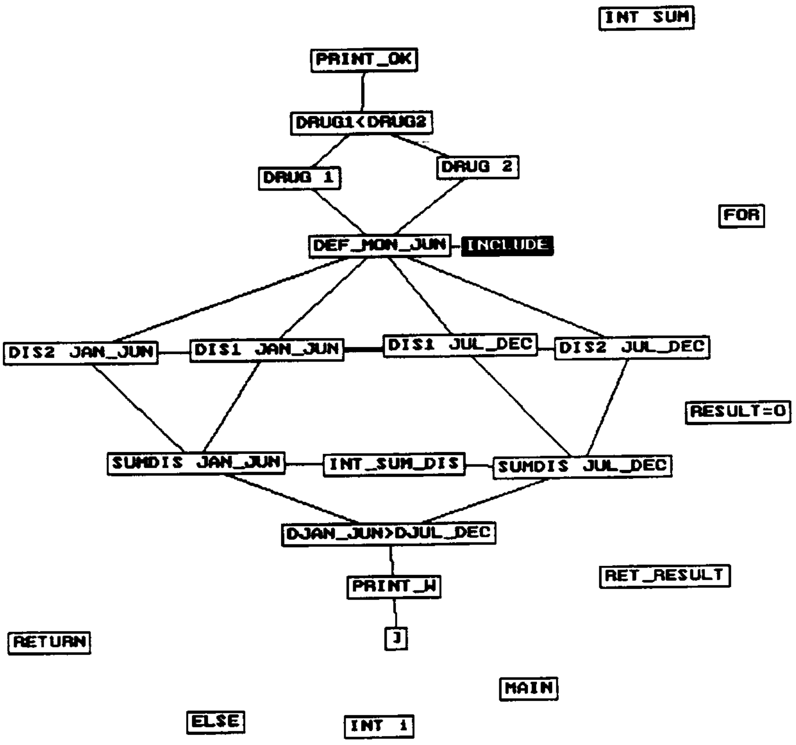
WIDOWSKY, D. & EYFERTH, K. (1986). Comprehending and recalling computer programs of different structural and semantic complexity by experts and novices. In H. P. WILLUMEIT, Ed. *Human Decision Making and Manual Control*, pp. 267–275. Amsterdam: Elsevier Science Publishers B. V., North-Holland.

WHITE, B. (1984). Design computer games to help physic students understand Newton's laws of motion. *Cognition and Instruction*, **1,** 69–108.

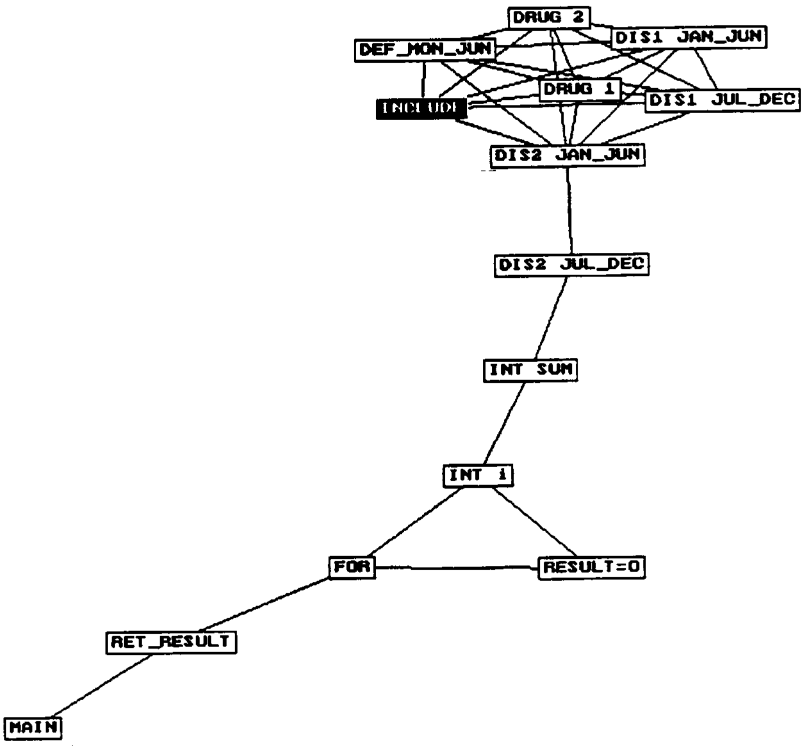
WHITLEY, K. N. (1997). Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, **8,** 109–142.

Paper accepted for publication by Associate Editor, Dr D. Boehm-Davis

Appendix A: Examples of the theoretical data flow: "drug problem" C version



Appendix B: Examples of the theoretical control flow networks: "drug problem" spreadsheet network


DRUG 2
DEF_MON_JUN
DIS1 JAN_JUN
DRUG 1
INCLUDE
DIS1 JUL_DEC
DIS2 JAN_JUN
DIS2 JUL_DEC
INT SUM
INT i
FOR
RESULT=0
RET_RESULT
MAIN