Restricting Manipulations Within a Device Space: Effects Upon Errors, Strategy and Display-Based Problem Solving

Simon P. Davies
Department of Psychology
University of Nottingham
University Park
Nottingham,
NG7 2RD
UK

Presented at the Fourth Psychology of Programming Interest Group Workshop, Loughborough University, January 1992.

Introduction

A pervasive finding of recent research into the cognitive aspects of programming is that code is not generated in a linear fashion - that is, in a strict first-to-last order (Davies, 1991; Rist, 1989). Typically, programmers make many deviations from linear development, leaving gaps in the emerging program to be filled in later. Hence, the final text order of the program rarely corresponds to its generative order. Green et al (1987) have proposed a model to account for this finding. Their parsing/gnisrap model describes the process by which a skeletal plan is instantiated in a programming notation This model introduces a working memory component into the analysis of coding behaviour which forces the model to use an external medium (eg the VDU screen) when program fragments are completed or when working memory is overloaded. In terms of this model, programs are not simply built up internally and then output to an external media with a generative order that reflects the final text order of the program.

However, the working memory limitations suggested by the parsing/gnisrap model give rise to other cognitive costs since programmers will frequently need to refer back to generated fragments in order to recreate the original plan structure which may have only been partially implemented in code. The parsing element of the model describes this process, while gnisrap (the reverse of parsing) describes the generative process. While the parsing/gnisrap model relies extensively upon the notion of working memory in order to explain the evident nonlinearities in program generation, it fails to address several key issues in relation to the role of working memory that have been raised elsewhere. One issue of particular importance is the relationship between working memory and the development of expertise.

One study that has considered this issue was conducted by Davies (1991). Davies looked at the nature of the nonlinearities found to exist in program generation for programmers of different skill levels. One finding to emerge from this work was that experts perform a greater number of between plan jumps than novices and that novices tend to perform more within plan jumps - that is, adopt a linear generation strategy. However, if we consider working memory to be a more flexible resource with a capacity that is related in part to skill development, we should expect the opposite result. In particular, if we assume that the re-parsing of a generated output involves some cognitive cost, then one might expect the development of programming skill to be partly dependent upon a programmer's ability to generate as much of the program internally before writing it to an external source, thus reducing the need to re-parse. However, the opposite appears to be the case. The results of Davies (1991) suggest that skilled programmers make much use of external memory sources (i.e., a VDU screen) while novices tend to rely upon the use of internal memory to develop as much of the solution as possible before transferring it to external memory. Moreover, novices rarely change their output once it is produced.

In light of the well documented relationship between working memory and the development of expertise that has been observed in other domains these findings are clearly rather anomalous. Given the cognitive costs that are involved in continually evaluating and modifying generated code, we require an explanation as to why skilled programmers rely so extensively on external rather than internal memory sources. One reason for this might be that programming demands the simultaneous assimilation of information from a range of problem space representations (Pennington, 1987). This integration of information is likely to place a significant load on working memory (Elio, 1986) and continually switching between different abstraction levels may incur too great a cognitive cost. This may give rise to the observation that

programmers develop code from focal structures, building the rest of the code around these fragments and using the external display as repository for intermediate solution steps.

Hence, the question that arises in the present context relates to the extent to which expertise in programming and in other complex skills can be explained by recourse to an extended working memory model as opposed to a model which places emphasis upon the role of externalised memory structures and display-based comprehension? The following experiments attempt to address this issue directly. The first experiment considers the role of working memory in the determination of strategy for novice and expert programmers. The second experiment looks at the effects upon certain error forms of restricting the kinds of manipulations programmers can make within an environment.

Experimental Studies

In the first experiment subjects were carried out an articulatory suppression task while engaged in a program generation activity. This experiment addresses a number of specific hypotheses. Firstly, if working memory limitations cause programmers to make use of an external medium, as suggested by Green et al, then the act of loading working memory through a concurrent task should give rise to an increase in nonlinearities. Given the effort required to use an external medium, in terms of the number of times a programmer must engage in the parsing/gnisrap cycle, one would expert experienced programmers to rely more extensively upon internal sources. Additional support for this hypotheses also arises from studies which suggest a strong link between expertise and working memory availability. However, the results of Davies (1991) give rise to an opposing hypothesis. This work suggests that skilled programmers make less use of internal sources than do novices and tend to rely more extensively upon using an external medium to record partial code fragments as they are generated. Hence, when working memory capacity is restricted this should give rise to a greater number of nonlinearities in the context of novice behaviour and only a small decrement in the case of experts.

The second experiment considers the role of working memory from a different perspective. Here interest is directed towards the way in which restricting the use of an external medium affects performance. In terms of the above analysis, if programmers are not able to correct already generated code at later stages in the coding process, then this should have some effect upon their performance. In this experiment, subjects created a program using a full-screen editor that provided no opportunity for the revision of existing text. The use of such an editor clearly places a significant load upon a subjects working memory capacity since they will be required to internally generate as much of the program as possible before externalising it. By placing emphasis upon the use of working memory it should be possible to induce error prone behaviour which parallels that evident when working memory is loaded in other ways, for instance via articulatory suppression.

We might therefore hypothesise that experts would perform worse than novices when the device used to create the program is restricted in such a way as to make retrospective changes impossible. This is based upon the assumption that experts make greater use of external sources to record partial code fragments that are then later elaborated through cycle of generative and evaluative activities.Conversely, it has been suggested that novices will tend to rely more upon generating as much of the program internally before writing it to an external source. It is clear that these strategic differences will be supported to a greater or a lesser extent by the device used to create the program. Hence, for expert programmers, it might be suggested that restricting the device will cause them to revert to a novice strategy, since they will then be unable to use the external display in the normal way.

Establishing support for this hypothesis would have a number of implications. Firstly, it would suggest that the development of expertise may not be based simply upon the acquisition of knowledge about a given domain. If this were the case, we would expect experts to perform better than novices regardless of the constraints imposed by the task environment. Secondly, it would indicate that increased working memory availability does not necessarily lead to better performance. Moreover, if increased working memory availability is correlated with expertise, then experts should perform better that novices in situations where they must rely upon internal sources. If this is not the case, then we might question the central status of working memory in theories dealing with the development of complex skills. An alternative explanation is to argue that experts have developed particular strategies for dealing with task complexity that involve close interaction with external information repositories in order to record partial solution fragments as they are generated. If novices have not developed such strategies, then it is unlikely that their performance would be affected significantly by restricting the task environment.

This analysis can be extended by classifying the errors in the programs generated by subjects. A scheme devised by Gilmore and Green (1988) suggests four main categories of error:

1 - Surface level errors caused mainly by typing and syntactic slips: (e.g. confusion between < and >, missing or misplaced quotes etc).
2 - Control-Flow errors: (e.g. missing or spurious else statements, split loops etc).
3 - Plan-Structure errors: Including, guard test on wrong variable, update wrong variable etc.
4 - Interaction errors: A class of errors occurring at the point where structures of different types interact: (e.g.a missing 'Read' in the main loop, initialisations within the main loop).

Clearly some of these errors will be knowledge-based (specifically, plan-structure errors) while others will be dependent upon working memory limitations. For example, both control-flow and interaction errors, since they depend upon establishing referential links and dependencies between code structures, are likely to be affected by working memory constraints. In terms of the first experiment, we might expect both control-flow and interaction errors to predominate in novice solutions where working memory availability is reduced. In the case of experts, it is argued that the interactions between code structures will be evaluated in the context of an external memory source. That is, by reparsing existing code fragments in order to reconcile them with the code the programmer is currently working on. Thus, that the act of loading working memory should not affect the occurrence of these types of error.

In the case of the second experiment we would expect the converse. If experts are not able to use the external display to aid problem solving in the manner predicted, then it might be hypothesised that interaction and control-flow errors will predominate in the condition where use of the device is restricted. It might also be predicted that this experimental manipulation will not affect the occurrence of plan-structure errors since these are hypothesised to be knowledge-based rather than strategy-based.

Experiment 1. Effects of articulatory suppression on programming strategy and errors

Method

Subjects
Twenty subjects participated in this experiment. One group of ten subjects consisted of professional programmers. All the subjects in this group used Pascal on a daily basis and all had substantial training in the use of this language. Members of this group were classified as experts. A second group consisted of second year undergraduate students all of whom had been formally instructed in Pascal syntax and language use during the first year of their course. Members of this group were classified as novices.

Materials and procedure
A variety of suppression tasks were explored, but the more complex tasks tended to disrupt performance to such an extent that a very simple articulatory suppression task was adopted. This involved asking the subjects to repeat a string of five auditorily presented random digits.The experimenter was present during the session in order to ensure that this concurrent task was performed, and intervened only when the subject paused for more than 5 sec. The subjects were requested to generate a simple Pascal program that could read a series of input values, calculate a running total, output an average value and stop given a specific terminating condition. This specification was derived from Johnson and Soloway (1985) and was chosen because it has formed the basis for many empirical studies and could be thus be more easily analysed for plan structures and errors. Subjects were allowed to study the specification for 5 mins and were then asked to generate a program corresponding to this specification while engaged in the concurrent suppression task. The subjects were given 15 mins. to complete this task, typing their solutions onto a familiar text editor. Subjects' keystrokes were recorded for further analysis. This analysis provided an indication of the temporal sequence in which programs were generated. Three independent raters were asked to analyse all the resulting program transcripts for the presence of common plan structures (Soloway and Ehrlich, 1984) and for errors (using the classification described above). Within and between-plan jumps were defined as follows: Within-plan jumps were classified as movements between a particular line of the program text to another line which formed part of the same plan structure. Between-plan jumps were defined as movements from the current line to lines within different plan structures (see Davies, 1991). These protocols applied only to situations where the jump was followed by an editing action.

The experiment was a two-factor design, with the following independent variables: 1. Articulatory suppression/No suppression and 2. Level of expertise (Novice/Expert).There were two dependent variables: 1. The number of Between/Within-plan jumps and 2. Errors remaining in the final program

Results

Plan-jumps

Figure 1 shows the number of within and between-plan jumps performed by novice and expert programmers in the two experimental conditions. Analysis revealed main effects of suppression ($F_{1,72}$ = 8.47, $p<0.01$) and expertise ($F_{1,72}$ = 12.56, $p<0.01$) on jump-type and a more complex interaction between suppression and expertise ($F_{1,54}$ = 4.73, $p<0.05$). A number of post-hoc comparisons were carried out using the Newman-Keules test with an adopted significance level of $p<0.01$. This procedure indicated that experts produced significantly more between plan jumps than novices in the non-suppression condition. Conversely, novices produced a greater number of within plan-jumps in this condition. In the case of the suppression condition, there were no significant differences.

Errors

Figure 2 shows the total mean number of errors remaining in the programs on task completion for novice and expert subjects in the two experimental conditions. Analysis revealed a main effect of expertise ($F_{1,36}$ = 9.37, $p<0.01$) and suppression ($F_{1,36}$ = 4.54, $p<0.05$) and an interaction between these two factors ($F_{1,36}$ = 15.89, $p<0.01$). Once again a number of post-hoc comparisons were carried out using the Newman-Keules test with an adopted significance level of $p<0.01$. This indicated a significant difference in error rates in the both experimental conditions when comparing the novice and expert groups. In addition, a significant difference between error rates across conditions was evident for the novice group. In the case of the expert group the same comparison proved not to be significant.

Error classification analysis

Figure 3 represents the proportion of errors in each error classification. In the case of experts, there is a fairly even distribution of error types across the two experimental conditions. Indeed, further statistical analysis revealed no significant differences between error types both within and between conditions (multiple t-tests). In the case of the novice group, the distribution of error types is less straightforward. In the non-suppression condition, novices produced a significantly greater number of plan errors in comparison to the other categories (t-test). Moreover, the only significant difference between the novice and experts groups in this condition was the number of plan errors produced by the novice group (t-test). In the second condition, the distribution of errors across classification types for expert subjects was again fairly even. No significant differences between any of the error classifications were evident (t-tests). For the novice group, significantly more control-flow and interaction errors were evident in comparison to the other two error classifications (t-test). Moreover, for the novice group,the number of plan errors occurring in the second condition was significantly less than in the first condition (t-test).

Discussion

This experiment shows that expert performance in programming tasks is not significantly affected by articulatory suppression. Hence, for experts the number of errors produced is not significantly different comparing the suppression condition to the non-suppression condition. Moreover, it appears that strategy is similarly unaffected. Hence, the prevalence of between-plan jumps in the non-suppression condition for the expert group is not diminished in the suppression condition. Similarly, the occurrence of within-plan jumps does not differ significantly in the two experimental conditions.

Conversely, the novice group produced significantly more errors in the suppression condition when compared to the non-suppression condition. In addition, the nature of the coding strategy that they adopt is also affected. In particular, it appears that novice programmers revert from a linear generation strategy characterised by the prevalence of within-plan jumps, to a strategy more characteristic of experts. That is, to a strategy which reflects a greater number of between-plan jumps.

Earlier it was stated that expert programmers appear to rely much more extensively than novices upon the use of external sources to record partial code fragments and that the act of loading working memory or of

otherwise reducing its availability would not affect this process. It was suggested that experts will tend engage in very closely linked cycles of planning, subsequent code generation and evaluation. Since it is posited that this process relies little upon the programmer's working memory capacity it is reasonable to expect that articulatory suppression would not affect the nature of performance in the context of this task. The results of this experiment provide support for this view. Further support for this view is evident in the error data. In the non-suppression condition, novice subjects are clearly more error prone than experts. This finding is not unexpected. However, in the suppression condition, the error rate for the expert group changes little from this base line whereas the novice error rate more than doubles. This may indicate that when working memory is loaded novices must externalise information and that this constitutes a strategy which they find unnatural, thus leading to an increased error rate.

A more detailed analysis of these errors reveals a change in the nature of errors for novice subjects between the two experimental conditions. In the non-suppression condition, the novice group make a greater number of plan errors, suggesting knowledge-based difficulties. Conversely, in the suppression condition a greater proportion of control-flow and interaction errors are evident. In terms of the present analysis, the preponderance of control-flow and interaction errors may reflect problems keeping track of the interdependencies between elements in the emerging program. When working memory availability is reduced it appears that novices experience some difficulty with these interdependencies. Unlike experts, it appears that novices cannot use the external display as an aid to memory to its full extent.

An alternative explanation for these findings is that experts simply have an extended working memory capacity. Such an account would presumably have no difficulty predicting the results of the experiment reported above. In order to assess the cogency of this alternative explanation, the second experiment reported in this paper adopts a different approach for exploring the relationship between working memory and the development of programming skill. In particular, if experts, for whatever reason, are able to extend their effective working memory capacity or increase its availability in other ways then restricting the task environment should not significantly affect their performance.

Experiment 2. Effects of restricting the task environment

The second experiment is complementary to the first. Whereas the first experiment attempted to reduce the subjects' available working memory capacity, this experiment has been designed to encourage subjects to rely upon working memory. Hence, if experts have an extended working memory capacity they should demonstrate performance equitable to that displayed in the first experiment. Moreover, if the extended capacity notion is correct, then experts should out perform novices even in the situation where the task environment is severely restricted as in this second experiment.

Method

Subjects

The same subjects took part in this experiment, with the order of participation randomised.

Materials and procedure

Subjects were asked to produce a program corresponding to a brief specification which involved processing simple bank transactions. Here, the nature of the task environment formed the basis for the two experimental conditions. In one condition, subjects used a familiar full-screen text editor. In the second condition subjects used a modified version of the same editor,which allowed only restricted cursor movement. That is, from the top of the screen to the bottom. and only between adjacent lines. Once a subject had generated a line and pressed the return key, they were unable to then return to that line to perform other editing operations. The editor did however allow edits to the current line being generated. Subjects first participated in a 5 min. familiarisation session, where the basic modifications to the editor were described. Subjects were then asked to attempt to generate a program from the specification and were asked to check each line of their program before pressing the return key, in order to determine whether they were satisfied with their response. 15 mins were allowed for this task.

Design

This experiment was a two-factor design with the following independent variables: Environment (restricted/unrestricted) and Level of expertise (Novice/Expert). In this case the dependent variable was the number of errors remaining in the final program.

Results

Errors

The results of this experiment are shown in figures 4 and 5. These data were analysed using a two-way analysis of variance with the following factors; Environment (restricted or unrestricted) and Level of expertise (Novice/Expert) This analysis revealed a main effect of Environment ($F_{1,36} = 5.74$, $p<0.05$), a main effect of Level of expertise ($F_{1,36} = 4.21$, $p<0.05$) and an interaction beween these two factors ($F_{1,36} = 9.76$, $p<0.01$). Post-hoc comparisions were carried out using the Newman-Keules test with an adopted significance level of $p<0.01$. This analysis revealed a significant difference between the number of errors produced by novices and experts in condition 1.

Error classification

The resulting program transcripts were analysed according to the classification scheme described above. The results of this analysis are shown in Figure 5. In the case of experts, analysis revealed no significant differences between error types within this condition (t-tests). In the case of the novice group, the distribution of error types in the first condition suggests a greater proportion of plan errors in comparison to the other categories (t-test). In the second condition, the distribution of errors across classification types for expert subjects was more complicated. This showed a greater proportion of control-flow and interaction errors compared to the other classifications (t-tests). In addition, experts produced significantly more control-flow and interaction errors in comparison to the first condition.

These results provide a striking demonstration of the effects of restricting a task environment. We have argued above that experts rely to a great extent upon using the external display to record fragments of code that are then further elaborated at subsequent points during the generation process. This led to the hypothesis that if programmers were unable to return to previously generated fragments then they would be forced into a situation where they would have to rely extensively upon working memory. However, it appears that while novices are seemingly unaffected by changes to the task environment, experts not only perform worse than novices but also produce the kinds of errors that are indicative of an inability to internally construct links and dependencies between code structures. These results reveal that experts produce more errors than novices in the restricted task environment. Moreover, experts produce a significantly greater number of control-flow and interaction errors in this second condition.

It was suggested previously that the first experiment that the results might be interpreted as indicating that experts have an extended working memory capacity. However, if this is the case then the results of this second experiment would appear to be rather anomalous. If we assume that experts have an extended working memory capacity in comparison to novices, then we might expect that situations which cause experts to rely upon working memory would not give rise to such an extensive decrement in performance. Moreover, in terms of this view there appears to be no reasonable explanation as to why experts produce many more control-flow and interaction errors in comparison to novices.

A more cogent explanation for these findings might simply involve suggesting that experts rely upon external sources and are not able to efficiently revert to a strategy that demands extensive reliance upon working memory. This would account for both sets of experimental findings. In the first experiment a reduction in working memory availability did not affect expert performance. This could clearly be accounted for in two ways. On the one hand, it could be argued that experts simply have an extended working memory capacity. Conversely, we might claim that experts rely extensively upon external sources and find it difficult to adopt other alternative strategies. However, the second experiment appears to suggest that the first of these explanations is incorrect. In particular, if experts have an extended working memory capacity then we would expect them to perform better than novices in situations where a reliance upon working memory is necessitated. This appears not to be the case.

Another finding relating to this data was that in the restricted environment condition the expert group produced fewer surface and plan errors. An explanation for this may be that, in the restricted environment condition, the normally automatic aspects of programming skill are disrupted. This may lead the programmer to attend to the knowledge-based components of programming skill leading to a reduction in surface and plan-based errors. There is evidence in the literature which suggests that so called 'skill' and 'knowledge-based' errors are to some extent disassociable (Reason, 1979).

Conclusions

These experiments have clearly demonstrated that the relationship between skill development in programming and working memory is not as predicted. It appears that experts rely significantly upon external sources to record code fragments as these are generated and then return later, in terms of the temporal sequence of program generation, to further elaborate these fragments. It has been suggested that a major determinant of expertise in programming may be related to the adoption or the development of strategies that facilitate the efficient use of external sources. The externalisation of information clearly has a high cost in terms of the reparsing or recomprehension of generated code that is implied. Hence, it might seem counterintuitive to suggest that problem solvers will tend to rely upon this kind of strategy rather than upon a strategy which involves the more extensive use of working memory. However, this explanation is consonant with existing work which has implicated display-based recognition skills in theoretical analyses of complex problem solving (Larkin, 1989). The contribution of these analyses has been important, but they have neglected to consider the relationship between display use and expertise and the consequent effect that this may have upon the nature of problem solving strategies.

The results of the experiments reported here question the cogency of accounts of expertise which place central emphasis upon the assumption that experts possess an extended working memory capacity or availability. For expert programmers, it has been shown that articulatory suppression affects neither programming strategy nor number of errors. Such an effect would be expected given the increased capacity assumption posited in previous models. However, the results of the second experiment would not be predicted on the basis of this assumption. In particular, restricting the task environment such that the programmer must rely more extensively upon working memory should affect neither strategy nor errors to the extent that was apparent in this particular study. Since it is uncommon to see expert performance reduced to level exhibited by novices, it might claimed that restricting the environment in this way causes experts to revert from their preferred strategy to one more characteristic of novices.

The work reported here also poses implications for the way in which we might attempt to explain the occurrence and distribution of error types. In particular, it is clear that a certain classes of error can be attributed to working memory limitations and that such errors are not distributed at random. In terms of the error classification employed here, it appears that interaction and control flow errors predominate in situations where working memory availability is reduced. Previous work (Anderson, 1989) suggests that errors arising from working memory failures will occur at random. However, the results of the studies presented here suggest that working memory related errors may have a more systematic distribution, and that the type of errors one might expect to occur may to some extent be predictable.

It also appears that that the nature of display-based problem solving in programming may be highly dependent upon features of the programming language considered. Green (1991) suggests that some programming languages are "viscous" in that they are highly resistant to local change. Hence, adding a line to a Basic program may involve renumbering lines such that the correct control flow is maintained. In terms of the present analysis, less viscous languages will provide better support for the kind of incremental problem-solving processes that are proposed. Such features will affect the strategies employed in the generation of code. However, there are other language features which will affect its comprehension. Gilmore and Green (1988) suggest that some languages are "role-expressive" (for example, Pascal) in that they may contain a rich source of lexical cues which enable a programmer to distinguish more easily the porgam's structure. They contest that less role-expressive languages (e.g. Prolog) are lexically amorphous and will tend not to facilitate certain forms of comprehension.

Such language features are important in the present context, since they will clearly affect the incremental nature of code generation and comprehension/recomprehension. This analysis extends existing work by suggesting ways in which language features and strategy may interact with features of the task environment to give rise to particular forms of behaviour. Such effects would not be taken into account by display-based views, since the salience of particular features of the display remains undifferentiated.

Summary

The experiments reported in this paper suggest that the development of expertise in programming is dependent upon the adoption of strategies for effectively utilising an external display. They also demonstrate that increased working memory capacity or availability is not a necessary prerequisite of skilled performance in this domain. Rather, skilled programmers appear to engage in closely linked cycles of code

generation and evaluation activities. According to this model, code is generated in a fragmentary fashion and the display is used as a repository for recording intermediate solution steps.

While this paper has indicated the importance of display-based performance in programming, it has also suggested two primary limitations of this general approach. Firstly, existing accounts of display-based problem solving ignore the apparent relationship between expertise and the development of strategies for utilising display-based information. Secondly, such accounts fail to consider the possibility that different forms of display-based information will be differentially salient in the context of a given task. Further developments of display-based accounts of problem solving will need to address these issues if they are to provide a coherent description of human performance in the context of complex tasks.

References

Anderson, J. R., (1983). *The architecture of cognition.* Harvard University Press, Harvard, MA.

Anderson, J. R., (1989). The anological origins of errors in problem solving. In D. Klahr and K. Kotovsky (Eds.), *Complex information processing: The impact of Herbert A. Simon.* LEA, Hillsdale, NJ.

Chase, W. G. and Simon, H. A., (1973), Perception in chess. *Cognitive Psychology*, **4**, 55-81.

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

Elio, R., (1986). Representation of similar well-learned cognitive procedures. *Cognitive Science*, **10**, 41-73.

Green, T. R. G., (1991). Describing information artifacts with cognitive dimensions and structure maps. In D. Diaper and N. Hammond (Eds.), People and Computers 6, Cambridge University Press.

Green, T. R. G., Bellamy, R. K. E. and Parker, J. M., (1987). Parsing and gnisrap: a model of device use, Proc. INTERACT'87, H. J. Bullinger and B. Shackel (Eds.), Elsevier Science Publishers B. V., North-Holland.

Johnson, W. L. and Soloway, E., (1985). PROUST: Knowledge-based program understanding. *IEEE Transactions on Software Engineering*, **SE-11**, (3), 423 - 442.

Larkin, J. H., (1989). Display-based problem solving. In D. Klahr and K. Kotovsky, (Eds.), *Complex Information Processing; The impact of Herbert A. Simon.*

Pennington, N., (1987). Stimulus structures and mental representation in expert comprehension of computer programs. *Cognitive psychology*, **19**, 295 - 341.

Soloway, E. and Ehrlich, K., (1984). Empirical studies of programming knowledge. *IEEE Trans. SE*, **SE - 10**(5), 595 -609.

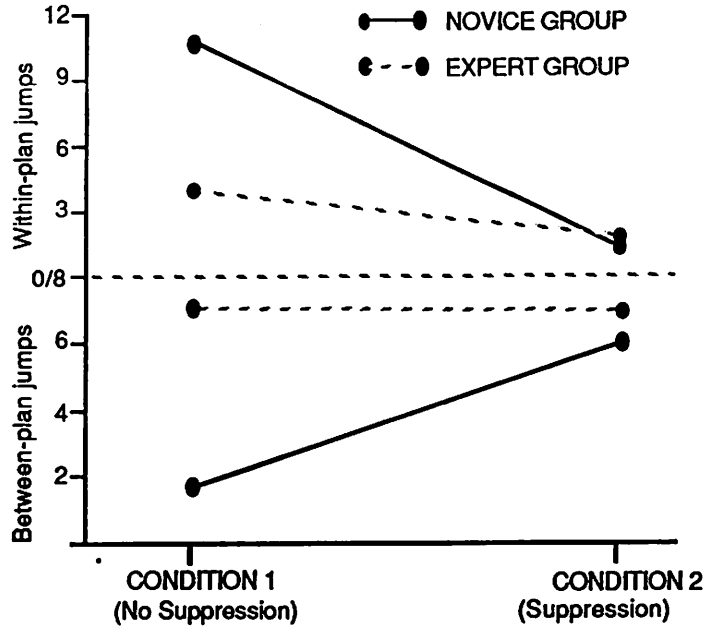

Figure 1 Within and Between-Plan jumps by novice and experts during the first experiment

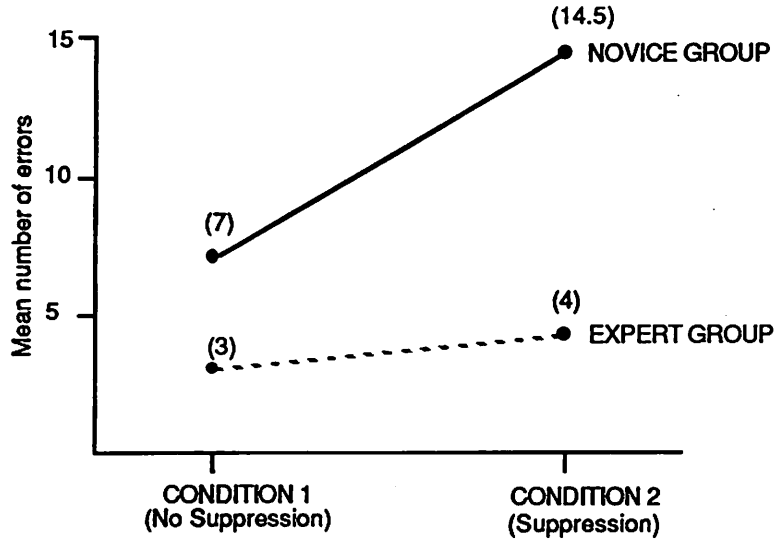

Figure 2 Mean number of error in experiment 1 for novice and expert subjects

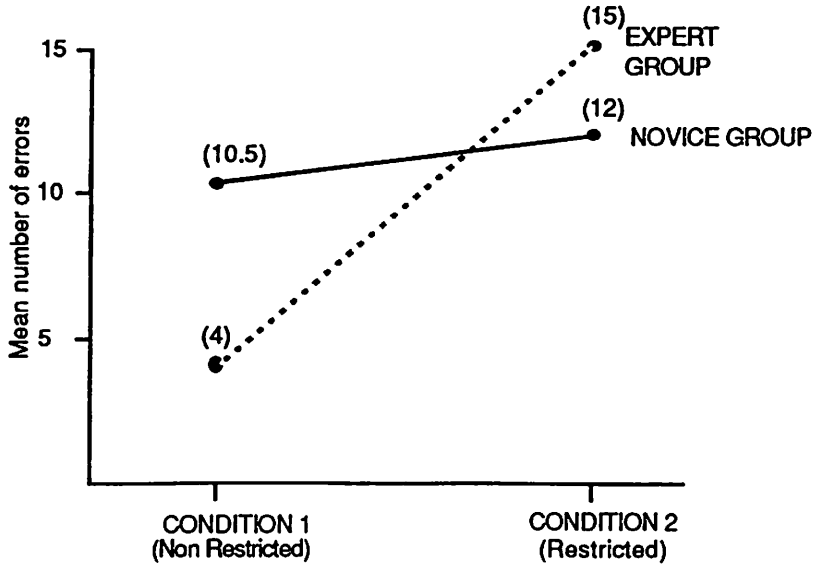

Figure 4 Mean number of errors in experiment 2 for novice and expert subjects

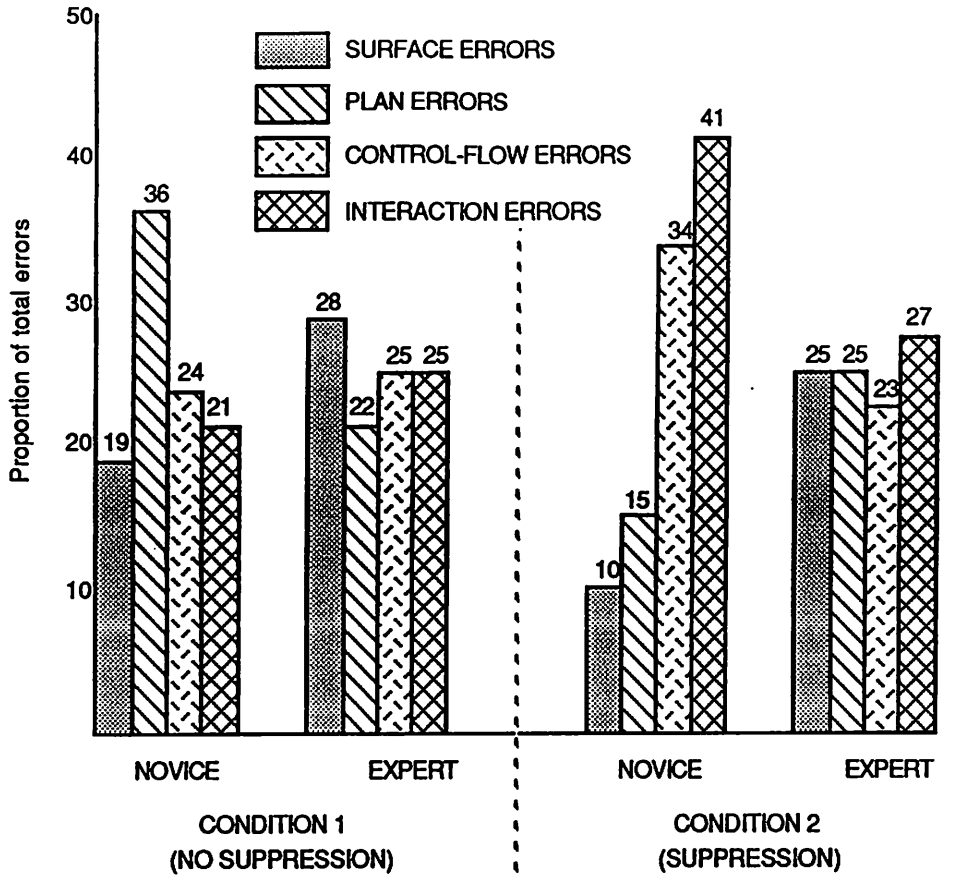

Figure 2. Errors on task completion in experiment 1

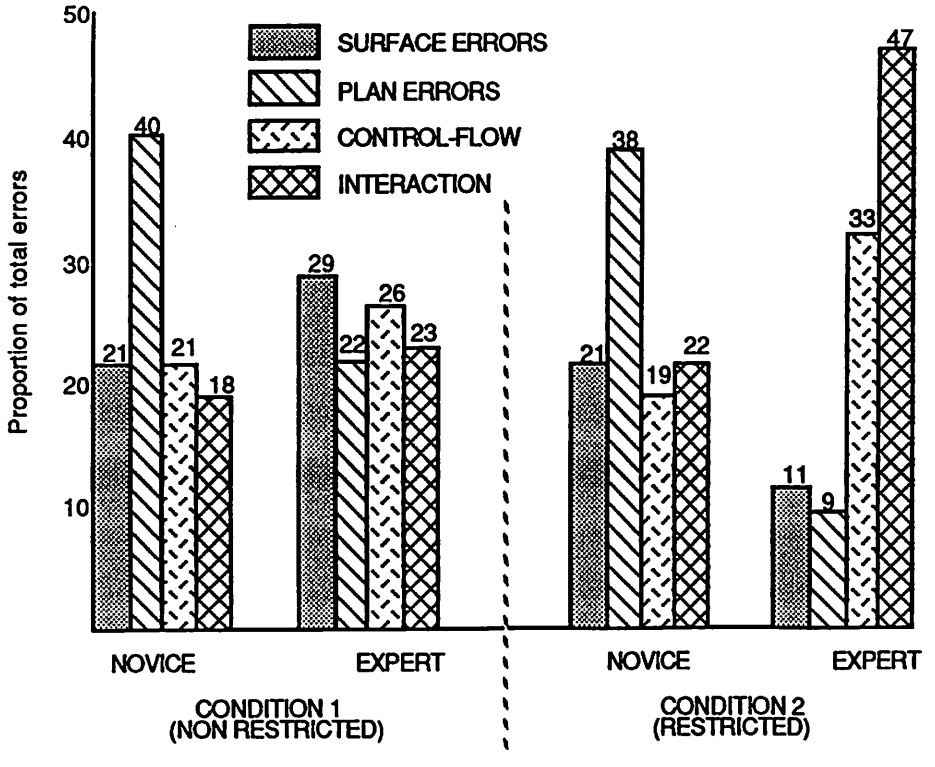

Figure 5. Errors on task completion in experiment 2