In P. Romero, J. Good, E. Acosta Chaparro & S. Bryant (Eds). Proc. PPIG 17 Pages 62 - 76

Graphical visualisations and debugging: a detailed process analysis

Pablo Romero, Benedict du Boulay, Richard Cox
Rudi Lutz and Sallyann Bryant

Department of Informatics, Sussex University, U.K.

Abstract. This paper investigates the question of how programmers exploit and integrate multiple sources of information. In particular it analyses how undergraduate computer science students used the multiple representations available in a software debugging environment (SDE). This environment allowed them to view the execution of a program in steps and provided them with concurrently displayed, adjacent, multiple and linked representations. These programming representations comprised the program code, two visualisations of it and its output.
This investigation studied debugging strategy in terms of rich process data about the use made of the representations available in the SDE and stepping facility. These data comprised computer interaction logs, audio recordings and data about visual attention focus.
The experimental results suggest that graphical representations seemed to promote a more efficient use of the available visualisations and were therefore associated with a relatively low level of interaction. This paper discusses these results and their implications for programming instruction.

1 Introduction

Much computer programming is performed via the use of software development environments which provide a variety of external representations and other sophisticated functionality. These representations and functionality enable programmers to treat programs not just as code text, but also as a range of abstract entities which can be visualised according to different criteria or executed under a variety of conditions. These visualisations can be presented in formats that range from mostly textual to mostly graphical [1].

The debugging step facility is one of the most helpful pieces of functionality of such environments. This facility allows programmers to execute and pause the program at different points. At these points they can inspect the visualisations provided to obtain information about various aspects of the program. Such program visualisation and debugging facilities should be especially helpful for novice programmers because they have the potential to enable them see the program not as a black box but as an abstract machine containing a set of elements and states. However, their effective use requires the programmer to deploy knowledge about how to decode and coordinate the available representations as well as skill in operating the SDE itself. It is often assumed that novices possess this knowledge. Thus, novice programmers can face a double challenge. As well as trying to learn abstract concepts about programming, they have to

master the decoding and representation coordination skills required to use debugging environments.

This study characterises the debugging strategies of Java novices in terms of step-and-trace choices and representation use in a multi-representational debugging environment. This characterisation is drawn from hybrid data; a mixture of data from different types but which originate from the same empirical episode. The types of data considered were computer interaction logs, audio recordings and data about visual attention focus. Section 2 explores research in programming strategy focusing on the way programmers manipulate the tools and representations available. Section 3 describes the experimental design and method. Section 4 presents the results of this study and Section 5 discusses these results. Finally, Section 6 presents some conclusions and describes further work.

2 External representation usage in programming

Programming studies have suggested that there are a range of strategies for finding errors in computer programs [2–5]. Bug finding strategies can be classified broadly into *forward reasoning* and *backward reasoning* [3]. The first category comprises those strategies in which programmers start searching for bugs from the program code, while the second involves starting from the incorrect behaviour of the program (typically its output) and reasoning backwards to the origin of the problem in the code. Examples of forward reasoning include *comprehension*, where bugs are found while the programmer is building a representation of the program and *hand simulation*, where programmers evaluate the code as if they were the computer. Backward reasoning includes strategies such as *simple mapping* and *causal reasoning*. In simple mapping the program's output points directly to the incorrect line of code, while in causal reasoning the search starts from the incorrect output going backwards towards the code segment that caused the bug.

Despite the debugging environment itself being a possible factor determining strategy, studies of debugging have mostly considered only the program code and its output as the two sources of information available for the debugging task. This is unfortunate because programmers normally work with a variety of visualisations and tools in addition to the program code and its output. Little is known about the way in which these visualisations and functionality affect strategy choice in debugging or about the way in which these resources are used in the debugging task.

Research on the use of multiple external representations in other areas has shown that an important factor when dealing with multi-representational systems is their heterogeneity in terms of modality. Here modality is used to mean the representational form used to present or display information, rather than in the psychological sense of a sensory channel. A typical modality distinction is between propositional and diagrammatic representations. It has been suggested that in general, the more graphically heterogeneous representations are, the more difficult it is for students to coordinate them [6].

A particular type of representation can constrain the interpretation of other representations because it might be more familiar or because of its inherent semantic properties. Graphical representations, for example, are weakly expressive (exhibit 'specificity') and

therefore compel the representation of specific information [7]. Such graphics can, in turn, constrain the interpretation of textual representations.

A series of studies of our own ([8], [9] and [10]) have investigated the issue of coordination of multiple external representations in program debugging. These studies tracked the visual attention of computer science undergraduates while debugging programs in a multi-representational (multi-window) debugging environment. We employed a modified version of the Restricted Focus Viewer (a visual attention tracking software system) to record moment-by-moment representation switching between concurrently displayed, adjacent representations. The studies suggested that there was a link between programming experience and a balanced use of the available representations. Although switches between the code and the visualisation were the most common, programming experience appeared to be associated with a more balanced switching behaviour between the main representation, the code, and the secondary ones. These studies also highlighted the importance of modality. In [9], for example, it was found that, in general, the higher the degree of interactivity with the SDE, the better the debugging performance. However, this relationship was not linear for those participants with a high level of skill working with graphical visualisations. The graphical condition seemed to promote a more efficient use of the available visualisations and was therefore associated with a relatively low level of interaction. However these studies only took into account data about representation usage. The study reported here builds on them but considers an additional type of data, the programmers' verbalisations, employing an analysis methodology that combines qualitative and quantitative aspects [11]. This study addresses the following questions in terms of the debugging behaviours and strategies deployed by programmers: i) Why is it that generally speaking the more interactivity with the system the better the debugging performance? and ii) Why was there a relatively low level of interaction with the visualisations (in the form of representation switching) for participants with a high level of debugging skill when considering modality?

3 Method

The main aim of the study described in this paper was to relate debugging behaviour (especially the use of the debugging step facility and the representations provided) and visualisation modality (employing either textual or diagrammatic visualisations) to debugging accuracy employing an analysis methodology that comprises qualitative and quantitative aspects. This detailed analysis explored participants' behaviour and debugging strategy by observing replays of the debugging sessions. This analysis took into account three types of data simultaneously: the trace of focus of attention, control of the presentation of the program's execution and verbalisation data. A detailed description of this methodology can be found in [11].

The detailed analysis took into account only a subset of the experimental data of the study reported in [9]. In this previous study, participants debugged six programs with one error each and employing either textual or graphical visualisations. They were encouraged to *think aloud* and their task was to identify the place and nature of the error as well as to suggest a fix for it. The detailed analysis reported in this paper focuses on

only one of the six target programs. The program version chosen was not significantly different to the other versions in terms of the use of representations (code and visualisations) that participants displayed and was the one that showed the widest spread of debugging accuracy scores. As one of the questions to address was related to representation modality, the subset of debugging sessions considered comprised examples of textual and graphical visualisation usage. This was a between factors experimental condition as some participants employed a version of the SDE with graphical visualisations and others with textual visualisations. Examples of these representations are presented in Figures 2 and 3. [1]

The SDE enabled participants to view the pre-computed execution of a Java program and presented, in addition to the code, its output and two visualisations of its execution. Participants were able to view the execution of the program by stepping between predefined *breakpoints* for a specific sample input. The SDE did not provide students with tools to edit, compile or re-execute the program with different input values or to reset breakpoints to places in the code. The motivation to limit the functionality of the tool in this way was to ensure, as much as possible, that all participants saw the same information and to reduce the complexity of operating the debugging environment.

Participants were able to see the program code, its output for a sample execution, and two visualisations of this execution. A screen shot of the system is shown in Figure 1. Participants were able to see the program class files in the code window, one at a time, through the use of the side-tabs. The *objects* and *call sequence* windows presented visualisations of the program's execution similar to those found in Object-Oriented software development environments [1]. The objects window (top right) presented data structure aspects while the call sequence window (bottom middle) showed control-flow information.

The SDE presents image stimuli in a blurred form. When the user clicks on an image, a section of it around the mouse pointer becomes focused. In this way, the program restricts how much of a stimulus can be seen clearly and thus indirectly allows visual attention to be tracked as the user moves an unblurred area around the screen. Use of the SDE enabled moment-by-moment representation switching between different program breakpoints and between concurrently displayed, adjacent representations to be captured for later analysis. The system was also able to record audio and to replay sessions, showing what participants did as well as what they said. In this way, the SDE can allow both quantitative and qualitative analyses of the recorded data. The user-computer interaction data (window and breakpoint fixation time and switching) can be analysed in a quantitative way (for example writing programs to process the logged data) to compare switching and fixation behaviour among the different experimental conditions. Observing replays of experimental sessions, on the other hand, can be used to interpret intentions and behaviours of participants. More details about the system and methodology employed can be found in [11].

Previous studies [8, 10] suggested that the restricted focus technology works best for program comprehension and debugging purposes if the unblurred area is of a size

[1] A detailed hybrid analysis such as the one performed is extremely time consuming and rather than focusing on a subset of the participants, we decided to select one (the same) debugging session for all of them

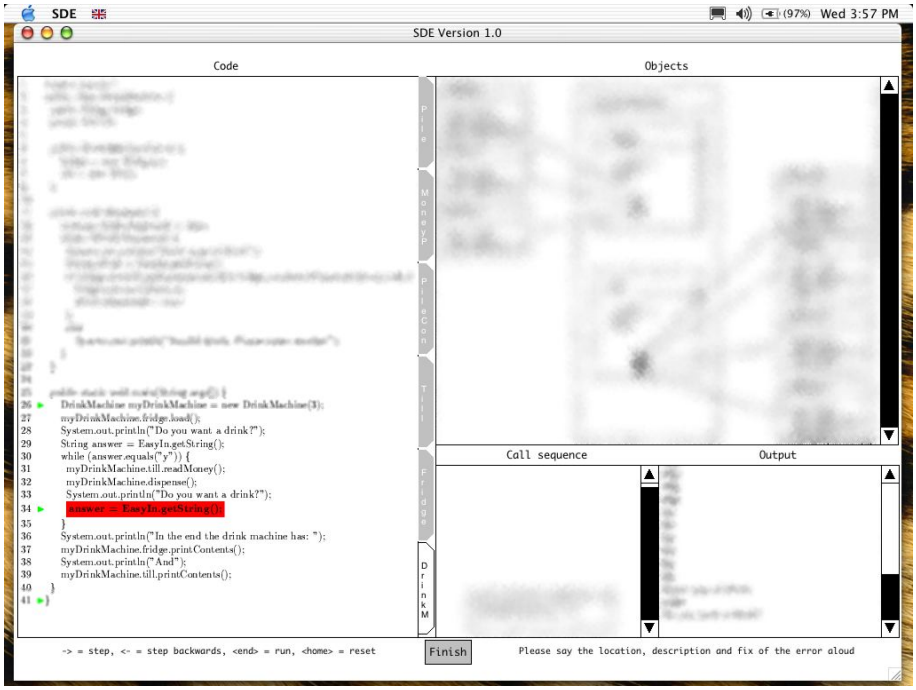

Fig. 1. The debugging environment used by participants

appropriate to cover entire representation units. In the case of the code, for example, these units can be equated to methods. The objects window represents an extreme case because the representation unit is the main object and therefore the unblurred spot covers the whole window. Studies that have validated the use of this technology have found that it does not modify task performance significantly [8, 12]. Studies that have compared visual attention behaviour using of this technology and employing eye-tracking equipment have however found differences in these two conditions [13]. The central issue concerns the validity of these techniques as measures of visual attention. Researchers have tended to interpret measurement differences between the two techniques as reflecting the superiority of eye tracking methods. However, recent evidence from the visual attention, change blindness and attention design literatures [14] questions this assumption.

3.1 Participants and procedure

The experimental participants were twenty-nine computer science undergraduate students from the School of Cognitive and Computing Sciences at Sussex University, U.K. All had taken a three month introductory course in Java. Some of them had previous programming experience, in most cases consisting of a few extra months of academic programming experience. Participants performed a program modification exercise, a

program comprehension activity and six debugging sessions. The experiment was divided into two sessions of about one hour each which took place on different days. The program modification and comprehension tasks were intended to familiarise participants with the program they were going to debug and with the program visualisations that were going to be presented to them in the objects and call sequence windows. Following the familiarisation tasks participants proceeded to the debugging part of the experiment.

The debugging session consisted of two phases. In the first phase participants were presented with samples of program output, both desired and actual. When participants were clear about the difference between these two sample outputs they moved on to the second phase of the session. In the second phase participants worked with the SDE. They were allowed up to ten minutes to debug the program and were instructed to think aloud and to identify the error in the program reporting it verbally by stating its location, description and a proposed fix for it.

The target program simulated the behaviour of a drink dispensing machine, was of medium size and complexity and was seeded with one error. This program loads the drink machine with cans of different drink types and also dispenses drinks after allowing the user to enter strings representing coins. The program is 201 lines long and comprises six classes linked by inheritance and composition relations. A typical execution of this program would create about 12 different objects, some of which are array data structures. The error in this program consisted in an incorrect initialisation of one array index and resulted in the amounts of each drink type being over-written to the same array location.

There were four predefined breakpoint lines in the code which generated six debugging steps or pauses. Sample objects visualisations for the textual and graphical conditions are shown in Figures 2 and 3 respectively.

A previous study [9] performed a quantitative analysis relating debugging accuracy to representation usage for the data of this experiment. That study divided participants on the basis of quartile ranges according to their debugging performance and found that, in general, the higher the degree of interactivity with the representations of the computerised environment, the better the performance. However, this relationship was not linear for those participants with the highest level of skill working with graphical visualisations. The graphical condition seemed to promote a more efficient use of the available visualisations and was therefore associated with a relatively low level of interaction. The qualitative analysis reported here takes the same definition of skill groups and tries to offer an explanation for these findings in terms of debugging behaviour and strategy.

4 Results

The detailed analysis quantified the frequency of occurrence of specific debugging behaviours. In order to perform this analysis, each debugging session was broken down into a sequence of discrete debugging events. These debugging events were related to the three types of data considered in this analysis: trace of focus of attention, control of the presentation of the program's execution and verbalisation data. A new event could

```
MyDrinkMachine :
    fridge :
        n_piles : 1
        piles :
            piles[0] :
                type : coke
                n_elements : 7
            piles[1] : null
            piles[2] : null
    till :
        n_piles : 4
        piles :
            piles[0] :
                type : 5p
                n_elements : 2
                value : 0.05
            piles[1] :
                type : 10p
                n_elements : 1
                value : 0.1
            piles[2] :
                type : 20p
                n_elements : 2
                value : 0.2
            piles[3] :
                type : 50p
                n_elements : 1
                value : 0.5
```

Fig. 2. Textual objects view of the *DrinkMachine* program.

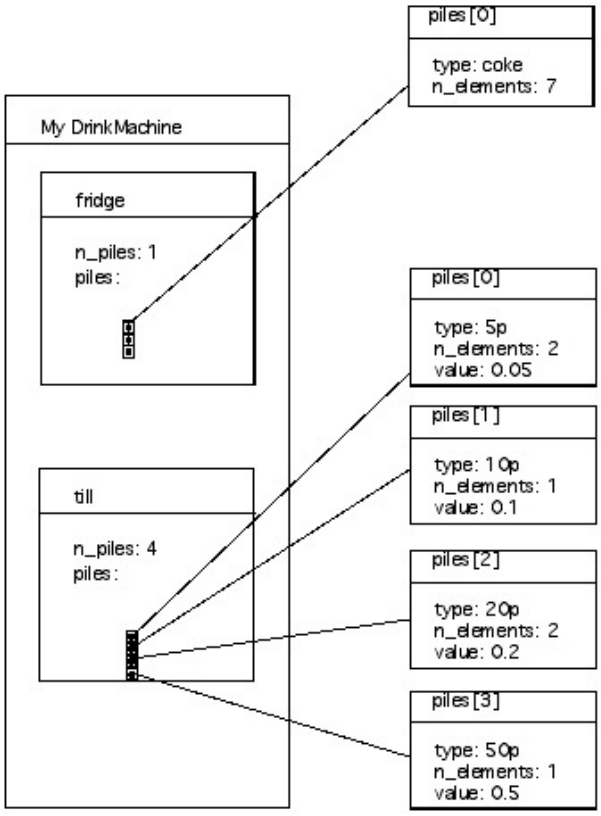

Fig. 3. Graphical objects view of the *DrinkMachine* program.

be triggered by a change in the focus of attention, a command related to the presentation of the program's execution, participants' verbalisations or a mixture of these. Therefore events were bounded by pauses or changes of topic in programmers' verbalisations (utterances), inter-window switches of visual attention focus or breakpoint switches. Each one of these debugging events was categorised as an instance of a specific debugging behaviour. The debugging behaviours taken into account are related to specific debugging strategies and are shown in Table 1. This categorisation was performed by replaying the debugging sessions and interpreting the three types of experimental data simultaneously. A detailed description of this methodology can be found in [11].

The categorisation of debugging events into the different debugging behaviours of Table 1 was performed by a rater who was trained to perform this interpretation but who

was unaware of the specific questions that this analysis was addressing (those at the end of Section 2).

Debugging behaviour	**Description**
Single stepping	Single stepping through the breakpoints of the program while watching either the objects or output view. Utterances describe the behaviour of the program in terms of the changes to its data structures or real world objects
Homing in	Homing in on an area of code after having uttered a debugging hypothesis. Participant is reading the code searching for the place responsible for the observed faulty behaviour
Code browsing	High level browsing of the code to build up a more complete picture of the program. Participant is reading the code for program comprehension purposes
Building a dynamic view	Talking about the program in terms of a dynamic view of it, but without stepping through it. Participant focuses on the code window commenting on dynamic aspects of the program without actually executing the program in steps
Comprehension hypothesis	The participant states a program comprehension hypothesis. This type of hypothesis is not directly related to the program error but rather to the way the program is implemented
Debugging hypothesis	The participant states a hypothesis regarding the identification of the error in the program

Table 1. Debugging behaviours taken into account in the detailed analysis of representation usage

Some of the debugging behaviours taken into account are related to the debugging strategies described in [3]. These debugging behaviours are described in Table 1. *Homing in*, for example, can be identified as the deployment of a *causal reasoning* strategy. *Code browsing* is associated to the *comprehension* strategy while *building a dynamic view* is related to *hand simulation*. There are not many references to a behaviour like *single stepping* in the debugging literature, perhaps because only a few debugging studies have taken into account the programmer's interaction with computerised debugging environments and in particular with the visualisations provided by them. *Single stepping* is the only behaviour, of those considered in this analysis, which makes specific reference to the available visualisations. *Comprehension hypothesis* and *debugging hypothesis* are not directly related to specific debugging strategies, rather, they can be considered as behaviours that may occur as part of different kinds of strategy.

Table 2 shows an example analysis sheet. This analysis sheet is produced by categorising the debugging session events as instances of debugging behaviours, as exemplified in Table 1.

In [9], participants were divided post-hoc on the basis of quartile ranges according to their debugging accuracy level. Quartile 1 comprised the participants with the

Time	Utterance	Focus of visual attention				Behaviour category
		Code	Objects	Call Sequence	Output	
0:29	Go, go, go. Come on. I see this.		piles[0] to piles[3]			Single stepping
0:43	Cool, I see this.		piles[0] to piles[3]			Single stepping
0:49	The problem is in the cans in the fridge, so...		piles[0] to piles[3]			Debugging hypothesis
1:01	Let's look at the fridge class	DrinkMachine. main (line 38)				Homing in
1:12		Fridge.load (line 16)				Homing in

Table 2. Section of analysis sheet for a specific debugging session

lowest scores while quartile 4 comprised those with the highest scores. In the analysis reported in this paper, this classification was taken as an indicator of individual ability and therefore as the basis to divide participants into different skill groups.

Table 3 illustrates the frequency of occurrence of specific debugging behaviours for each skill group. This frequency of occurrence was normalised, so the values reported are number of occurrences per minute. These figures show trends rather than statistically significant results, as the experimental variables taken into account did not exhibit a normal distribution, probably due to the low numbers of participants in each condition. This table suggests that groups of low skill (1 and 2) performed less single stepping than those of high skill (3 and 4).

	Group 1	Group 2	Group 3	Group 4
Single stepping	1.80	1.61	4.00	2.91
Homing in	0.95	0.49	0.74	1.44
Code browsing	3.56	3.70	3.62	2.46
Dynamic view	0.22	0.00	0.00	0.00
Comprehension hypothesis	0.20	0.23	0.16	0.52
Debugging Hypothesis	0.57	0.39	0.53	0.85

Table 3. Mean frequencies of debugging events (per minute) by skill group.

Low groups focused mainly on the code and therefore tended to do more code browsing. Also, it seems that their homing in episodes were supported by the code rather than by the visualisations. Their behaviour was guided by a mixture of comprehension debugging strategies (as defined in Section 2) as well as by genuinely trying to understand the program rather than attempting to debug it. In this latter case they were concerned with understanding the code, frequently at the level of decoding syntax (*'is this a method? It is, isn't it?'*). Sometimes they did not succeed at this task (*'I'm just looking at all the different methods. I'm not really seeing them though, just seeing lots of meaningless words'*). When they were employing a comprehension debugging strategy, the types of hypotheses they uttered were frequently syntax-related (*'it might just have too many closed brackets, I think line 41 has too many closed brackets'*). In both cases episodes of reading code aloud were frequent (*'class MoneyPile... n dot elements times value'*). Sometimes they were able to extract the relevant information from the available representations. However, big gaps in their knowledge prevented some of them from understanding important aspects of the program and led them to infer the wrong conclusion (e.g. *'That constructor class would have to make a type of drink rather than just a string'* [2]).

Group 3 performed the highest rate of single stepping. This tendency suggests that they had a high level of interaction with the visualisation windows. Their protocols frequently show detailed descriptions of the visualisations (*'Five cokes are entered. Number of cokes, number of drinks... right, is it?'*). It seems that their global debugging strategy combined episodes of looking at the execution of the program in steps to find out what was wrong with it and then trying to reason backwards to find the place in the code were the error was generated. However, sometimes gaps in their programming knowledge got in their way (*'Protected, protected, protected. Well it can't be because... they are all protected... I forgot what that means'*). Despite these problems, their behaviour is different from participants of the two lower skill groups in that they were trying to do debugging rather than program comprehension. They were looking at the visualisations trying to extract information about the program error and then referring back to the code attempting to find the place responsible for this error.

Participants of Group 4 did less *single stepping* than those of Group 3. However they did the most *homing in* of all groups. They proceeded in much the same way as participants of Group 3, combining periods of *single stepping* and *homing in* behaviours, apparently deploying a backward reasoning strategy to debug the program. However their presumably more advanced programming knowledge enabled them to be more successful when trying to find the place in the code responsible for the error. Associated with this success is the fact that they produced the highest number of hypotheses. They also performed the least amount of code browsing of the four groups. This fact strenghtens the conclusion that their behaviour was guided mainly by a backward reasoning strategy.

The second research question of Section 2 involved the comparison of behaviours for graphical and textual conditions for members of the high skill groups. Due to the low numbers of participants in these skill groups for these conditions (for example,

[2] In this case the participant is suggesting that there is an error with the type of object returned by a constructor. An error of this type would show at compilation rather than at execution time

for Group 4, three participants in the textual and four in the graphical condition), any comparisons have to be considered with caution. Table 4 presents the frequency of occurrence of debugging behaviours for the members of groups 3 and 4 taking modality into account. This table shows that members of Group 4 had a low frequency of single stepping but a high frequency of homing in behaviours in the graphical condition. They also produced a high number of debugging hypotheses.

Participants of Group 4 in the textual condition were similar to those of Group 3 in that sometimes they described the visualisations in a detailed way. Also, they sometimes viewed the same execution episode in steps several times (*'Let's go back to the start. Selecting the piles. Hmm, the next section has added coke to the first element of the pile... Returned 5 cokes. But for some reason it's added three...*'). Participants in the graphical condition, on the other hand, seemed to report the relevant information in the visualisations in a more direct way and then moved on to try to identify the place in the code responsible for the observed behaviour (*'Ok, I see this. Cool, I see this. The problem is the cans in the fridge, so ... Let's look at the fridge class'*).

	Group 3		**Group 4**	
	Text	Graphic	Text	Graphic
Single stepping	3.21	4.79	6.17	1.6
Homing in	0.67	0.81	0.31	1.90
Code browsing	4.09	3.16	2.78	2.32
Dynamic view	0.00	0.00	0.00	0.00
Comprehension hypothesis	0.18	0.15	0.22	0.63
Debugging Hypothesis	0.50	0.56	0.41	1.02

Table 4. Mean frequencies of debugging events (per minute) for Groups 3 and 4 by modality.

5 Discussion

This section discusses to what degree the results of the detailed analysis described in this paper can explain the results of our previous study [9]. In this study it was found that, in general, the higher the degree of interactivity with the SDE, the better the debugging performance. However, this relationship was not linear for those participants with the highest level of skill working with graphical visualisations. The graphical condition seemed to promote a more efficient use of the available visualisations and was therefore associated with a relatively low level of interaction.

The results of the analysis described in this paper, which took the same data and skill groups, suggested that varying degrees of interactivity can be explained in terms of the debugging strategies participants were deploying. Participants displaying a single-stepping behaviour would interact much more with the visualisations (and therefore with the SDE) than those that were, say, browsing the code or hand-simulating it. Participants of Group 3 were the ones with the highest frequency of single-stepping, and

this can explain why this group was also the one with the highest degree of interactivity with the SDE in terms of switches of visual attention between the representations.

The results of this study regarding the difference between graphical and textual conditions need to be considered with caution due to the low numbers of participants taken into account for this comparison. However results suggest that the relative low level of interactivity for the graphical condition of Group 4 could be explained by the low frequency of single-stepping performed in this condition. Participants in this condition also exhibited a high frequency of homing-in behaviours and produced a high number of debugging hypotheses. This difference in behaviour might be related to the different support given by the visualisations in each modality condition. Textual representations seemed to be more difficult to decode and interpret than graphical ones. They required participants to devote more time to understand the dynamic aspects of the program, as they had to identify relevant elements of the visualisation and then combine these elements into meaningful structures. Graphical representations, on the other hand, seemed to enable a faster, more direct understanding of these structures and allowed participants to move on to construct hypotheses about the program error and to verify these hypotheses against the program code. A comparison between Figures 2 and 3 illustrates this difference. Both figures encode the same information. However by grouping certain elements in boxes, Figure 3 helps to identify meaningful structures in the visualisation (in this case the objects of the program execution). Participants working in the textual condition, on the other hand, had to perform this grouping and then keep a reference to these meaningful structures in working memory. These processing overheads can be crucial when dealing with dynamic representations, as participants also had to detect patterns of change through time in the visualisations. This meant that participants in the textual condition probably tried to keep a reference to the meaningful structures of a series of visualisations. Cognitive overload might have caused these participants to view the same execution episode several times. These results seem to confirm the view that diagrams, unlike propositional representations, exploit perceptual processes by grouping relevant information together and therefore make the search and recognition of information easier [15].

Other studies have also highlighted differences in representation switching patterns between participants with different levels of skill. In [16] and [17], for example, it was reported that poor performers switched more frequently than successful ones in analytical reasoning tasks. However, there are several differences between those studies and the one reported here. Although analytical reasoning as a cognitive task might be remarkably similar to program comprehension, the analytical reasoning studies encouraged participants to build their own representations. Therefore, switching representations represented ‘a strategic decision by the subject to abandon the current external representation and construct a new one’ [17]. In the present study, representations were complementary (and pre-constructed) rather than alternative, therefore, switching did not necessarily represent discarding one representation for another, but more likely complementing the information of one with another. The reason for switching in the present study had more to do with cross checking between visualisations and possibly with an inefficient use of the visualisations, rather than with giving up on specific representations.

The main behavioural difference between groups of low and high levels of debugging accuracy lay in the strategy they deployed. Low debugging accuracy groups primarily performed a mixture of program understanding and comprehension, a forward reasoning debugging strategy discussed in Section 2. In the former case they were concerned with trying to interpret the program code and rarely attempted to decode the visualisations. This behaviour is understandable, as the code is the main representation of the program and serious problems in its understanding would very likely make any error finding activities unsuccessful. Higher levels of debugging accuracy were associated with a more frequent use of the available visualisations. The main way to perform these activities was by trying to spot manifestations of the error in the available representations and then attempting to identify the place in the code responsible for these manifestations. Sometimes gaps in their programming knowledge prevented participants from identifying the faulty code segment, and this section has already discussed the way in which properties of the available visualisations could make the task of spotting error manifestations in the available visualisations more or less difficult.

Analysing the results of this study together with those of a previous one that focused only on representation usage ([9]), it can be said that in general, debugging success was associated with following a debugging strategy which included viewing the execution of the program in steps. Participants who exhibited this behaviour interacted frequently with the program visualisations to complement the information in the program's code. On the other hand, poor debugging accuracy was associated with browsing the program code without referring to any of the other representations, trying to debug the program while building a representation of it. This behaviour was associated with a low frequency of interaction with the program visualisations. Additionally, participants with high levels of debugging accuracy displayed different behaviours when working with textual and graphical representations. The graphical modality was associated with a more efficient use of the visualisations. Therefore, participants working under this condition needed to interact less frequently with the visualisations to obtain information relevant for finding the error in the code.

These differences in debugging behaviour suggest different learning needs. According to the results of this study, students with a low level of programming skill might not benefit from learning activities which involve the need for coordinating several representations; it might be better for this group of programmers to work primarily with one representation with the aim of familiarising them with its formalisms and improving their decoding skills. Students with a higher level of skill, on the other hand, might benefit from learning tasks involving other representations in addition to the program code, especially if explicit instruction about the format of these additional representations and the relationship between them is given.

The main results of this study suggest that, at least for the experimental conditions considered, graphical representations enabled a more direct understanding of the relevant structures in the problem space. However this does not mean that diagrams are superior to textual representations for every situation, or that they will provide a good level of support in all cases. One of the main issues to consider is scalability. Programs, even for small projects, very often involve dozens of objects. Presenting all of them on

the screen can create layout difficulties for the designer of such a tool and probably cognitive overload problems for its users.

6 Conclusions

This paper has reported a study investigating the use of dynamic multiple external representations for program debugging by programming students exhibiting various levels of debugging accuracy. Representation usage was measured via a debugging environment complemented with functionality to track participants' visual attention and to record interaction with the system as well as their verbalisations. An analysis that took into account these three sources of information found that programmers with a low level of debugging accuracy have difficulties in interpreting the information in the program code and therefore resort to program debugging strategies that focus mainly on the program code when performing debugging tasks. Programmers with a higher level of accuracy, on the other hand, deploy a backward reasoning strategy which tries to identify manifestations of the error in the visualisations provided and also attempts to locate the places in the code responsible for these manifestations. Graphical visualisations seem to promote a more efficient use of the available visualisations for this group of programmers.

These results suggest that programmers of different levels of skill have different educational needs regarding representation coordination. Those of low skill levels may benefit from learning activities aimed to master the syntax and semantics of one representation at a time (particularly of the program code). Those of higher skill levels, on the other hand, may benefit from activities requiring the coordination of the program code and additional program visualisations, as well as from receiving explicit instruction about their format and interrelationships.

Acknowledgments

This work was supported by the EPSRC grant GR/N64199 and the Nuffield Foundation grant URB/01703/G. The support for Richard Cox of the Leverhulme Foundation (Leverhulme Trust Fellowship G/2/RFG/2001/0117) and the British Academy is gratefully acknowledged. The authors wish to thank Stephen Grant for his hard work both in refining the coding categories of the detailed analysis of representation usage and in the coding tasks of this analysis.

References

1. Romero, P., Cox, R., du Boulay, B., Lutz, R.: A survey of representations employed in object-oriented programming environments. Journal of Visual Languages and Computing **14** (2003) 387–419
2. Jeffries, R.: A comparison of the debugging behaviour of expert and novice programmers. In: Proceedings of AERA annual meeting. (1982)
3. Katz, I., Anderson, J.R.: Debugging: an analysis of bug location strategies. Human-Computer Interaction **3** (1988) 359–399

4. Vessey, I.: Toward a theory of computer program bugs: an empirical test. International Journal of Man-Machine Studies **30** (1989) 23–46
5. Gilmore, D.J.: Models of debugging. Acta psychologica **78** (1991) 151–172
6. Ainsworth, S., Wood, D., Bibby, P.: Co-ordinating multiple representations in computer based learning environments. In Brna, P., Paiva, A., Self, J., eds.: Proceedings of the 1996 European Conference on Artificial Intelligence on Education, Lisbon, Portugal (1996) 336–342
7. Stenning, K., Oberlander, J.: A cognitive theory of graphical and linguistic reasoning: logic and implementation. Cognitive Science **19** (1995) 97–140
8. Romero, P., Cox, R., du Boulay, B., Lutz, R.: Visual attention and representation switching during java program debugging: A study using the restricted focus viewer. In Hegarty, M., Meyer, B., Narayanan, N.H., eds.: Diagrammatic Representation and Inference. Second International Conference, Diagrams 2002. Lecture Notes in Artificial Intelligence 2317. (2002) 221–235
9. Romero, P., du Boulay, B., Lutz, R., Cox, R.: The effects of graphical and textual visualisations in multi-representational debugging environments. In Hosking, J., Cox, P., eds.: 2003 IEEE Symposium on Human Centric Computing Languages and Environments. IEEE Computer Society, Auckland, New Zealand (2003) 236–238
10. Romero, P., Lutz, R., Cox, R., du Boulay, B.: Co-ordination of multiple external representations during java program debugging. In Wiedenbeck, S., Petre, M., eds.: 2002 IEEE Symposia on Human Centric Computing Languages and Environments. IEEE press, Airlington, Virginia, USA (2002) 207–214
11. Romero, P., du Boulay, B., Cox, R., Lutz, R., Bryant, S.: Dynamic rich-data capture and analysis of debugging processes. In Dunican, E., Green, T., eds.: Proceedings of the 16th annual workshop of the Psychology of Programming Interest Group. (2004) 140–150
12. Jansen, A.R., Blackwell, A.F., Marriott, K.: A tool for tracking visual attention: The restricted focus viewer. Behavior Research Methods, Instruments, & Computers **35** (2003) 57–69
13. Bednarik, R., Tukiainen, M.: Visual attention and representation switching in Java program debugging: a study using eye-movement tracking. In Dunican, E., Green, T., eds.: Proceedings of the 16th annual workshop of the Psychology of Programming Interest Group. (2004) 159–169
14. Wood, S., Cox, R., Cheng, P.: Designing for attention: Eight issues to consider. Computers in Human Behavior **22** (2006)
15. Larkin, J.H., Simon, H.A.: Why a diagram is (sometimes) worth ten thousand words. Cognitive Science **11** (1987) 65–100
16. Cox, R.: Representation interpretation versus representation construction: a controlled study using switchERII. In du Boulay, B., Mizoguchi, R., eds.: Artificial intelligence in education: knowledge and media in learning systems (Proceedings of the 8th. World Conference of the Artificial Intelligence in Education Society, Amsterdam, IOS (1997) 434–444
17. Cox, R., Brna, P.: Supporting the use of external representations in problem solving: The need for flexible learning environments. Journal of Artificial Intelligence in Education **6** (1995) 239–302