Some Thoughts on Designing an Intelligent System for Discovery Programming

Haider Ramadhan
Computer Science and Artificial Intelligence
School of Cognitive and Computing Sciences
University of Sussex,
Brighton, BN1 9QH, UK.

e-mail: haiderr@cogs.susx.ac.uk

Abstract

This paper highlights and discusses some important design principles and issues for developing an intelligent system for discovery-oriented programming. The envisioned system synthesizes features of Human Computer Interface (HCI) with features of an Intelligent Tutoring System (ITS). In terms of HCI, the system is capable of providing novices with an open-ended, exploratory, and free discovery programming environment (microworld) that enables them to observe and discover the dynamic behavior of both individual elementary programming concepts and whole programs, and thus build the underlying conceptual knowledge associated with these concepts and a mental model of the programs' execution. In terms of an ITS, the system is capable of automatically analyzing and debugging novices' partial solutions for semantic errors during a guided discovery programming phase and provides them with intelligent feedback that guides them in the problem solving process.

Keywords

Intelligent tutoring systems, microworlds, discovery systems, automatic debugging systems, visual programming, program visualization, case-based reasoning and immediate feedback.

1 Introduction

Programming is a ubiquitous and cognitively demanding task. Novice programmers have difficulties in learning to program [Pea 1986, Bonar 1985, Soloway 1984, Anderson 1982, du Boulay 1986, Eisenstadt 1991]. Some computer scientists even say that computer programming is just too hard and too mathematical for novice programmers [Dijkstra 1982].

There are two main reasons behind these difficulties: lack of programming knowledge and lack of programming experience. Programming knowledge deals with the exact syntax and semantics of the programming language constructs being learned. This includes understanding the dynamic behavior of programming concepts such as variable declaration and binding, input and output operations, conditional statements, looping constructs and other more abstract concepts such as recursion. Programming experience deals with the skill required to connect the low-level syntax and semantics of constructs to produce properly integrated higher-level plans (programs and algorithms) [Bonar 1985, Chi 1982]. In other words, programming experience deals with the ability to put individual programming concepts together to come up with a complete solution for a given problem. This requires acquiring a mental model of how the computer executes the program, so that the reasoning through this execution can become possible.

This paper discusses principle design issues and decisions for developing an intelligent discovery programming system. The envisioned system helps novices to acquire both programming knowledge and programming skill. This is accomplished in two phases:

- In the first phase, the free discovery programming phase, the system helps novices observe and discover the dynamic behavior of individual programming concepts as well as whole programs to build the underlying conceptual programming knowledge needed for problem solving tasks without requiring them to mentally simulate the intricate behavior of the machine.
- In the second phase, the guided discovery programming phase, novices compose and coordinate programming concepts and language constructs, discovered and observed in the first phase, to solve given problems under the intelligent guidance of DISCOVER, and thus transform their programming knowledge into programming skill.

2 Main Design Issues

2.1 An Integrated System Image

A discovery programming system encourages a novice programmer to become an active learner by allowing him to form his own hypotheses, explore his own questions, and draw his own conclusions. The system develops the novice's

programming knowledge as an opportunistic learner by providing him with an exploration-based, free discovery programming environment in which he can explore programming concepts, discover their dynamic behavior through observation, detect any misconceptions associated with them, and hence build the necessary underlying conceptual programming knowledge. To achieve these goals, a discovery programming system needs to support visibility [du Boulay 1981], program visualization [Myers 1988], and a concrete model of the underlying computer system with which the novice is interacting [Mayer 1981]. du Boulay calls this concrete model of the machine the 'notional machine', while Norman [1983] calls it the 'system image'.

Mayer [1981], Norman [1983], Jones [1983], and Mayer [1989] suggest that novices learn the concepts of a programming language more effectively and more easily if they are presented with a concrete model of the underlying computing machine. Similar results have been reported by Olson [1986] who suggests that the main difficulty novices confront when learning to program is the assimilation of a model of the computing machine. These findings imply that the clarification of the notional machine facilitates the task of learning to program for novices. However, Jones also reports that providing a static concrete notional machine on its own is not enough and that even when presented with a notional machine, novices tend to build inaccurate models of programs' execution and still have difficulties in comprehending the flow of control in such dynamic concepts as looping constructs, conditional statements and procedure calls. Therefore, a discovery programming system not only needs to present novices with a notional machine but also needs to support program visualization and visibility to allow them visualize how their programs dynamically behave and how hidden and internal changes in the notional machine take place, and thus build a robust model of the program's execution and the machine's behavior. This includes, for example, seeing how variables are named, how they get their values, how the corresponding memory cells in the memory space are affected and how the control flows from one statement to another.

Most existing programming tutors and systems present novice programmers with a static view of the program's execution and its dynamic behavior, ignoring the issue of providing a visible, graphic and concrete base on which novices may build their underlying conceptual programming knowledge and programming skill. Some examples of these systems include Proust [Johnson 1984], Talus [Murray 1985], Aurac [Hasemer 1983], Laura [Adam and Laurent 1980], Spade [Miller 1982] and the Lisp Tutor [Anderson 1985]. As a consequence, novices are required to mentally simulate the execution of the program which they are writing and imagine its dynamic behavior and side-effects: a task they normally fail to accomplish.

Several programming systems have attempted to incorporate some of these design features in their implementation. However, these systems support only one or two of these features and lack an *integrated image* of the notional machine: the incorporation of visibility and program visualization within a concrete model

of the underlying computer system. Therefore, they fall short of providing a true discovery programming system or environment. Bip [Barr 1976], an ITS for BASIC, through showing pointers which move around the program code as it is executed and changes in the values of variables, supports program visualization and simple visibility. Similar features were also provided by programming systems for FORTRAN [Shapiro 1974], for PASCAL [Nievergelt 1978], and for assembly languages [Schweppe 1973, Shapiro 1974]. Bridge [Bonar 1988] also supports program visualization by highlighting program lines during execution and showing how different programming plans are connected in a graphic and diagrammatic fashion. By supporting the visual execution of whole programs, it becomes possible for these systems to help novices build a mental model of the whole program's execution. However, they lack a truly interactive environment that allows novices not only visualize the dynamic behavior of whole programs but also visualize the behavior of individual programming concepts and language constructs.

Visual Programming systems, such as BALSA [Brown 1986] and Tinker [Lieberman 1985], also attempt to show novices the dynamic behavior of the whole programs and algorithms. However, these systems emphasize the representation of programs in graphical terms, sometimes in more than two dimensions. There is no empirical evidence yet to suggest that visual programming is inherently more effective than conventional linear and text-oriented programming for teaching novices how to program. In fact, as Green [1990] claims, instead of focusing on visual programming and creating new programming notations, one can continue with the existing notation but use enhanced typography to make perceptual cues reflect the notational structure.

2.2 Case-based Reasoning

It has been well advocated even before Socrates' time that people reason about the situations they find themselves in by referring to similar situations that they have experienced, heard or seen. Therefore, a discovery programming system should develop the novice's programming capabilities as a case-based learner by providing him with relevant cases (examples) to help him in tackling his own programming problems. This is different from learning-by-analogy which focuses on issues of analogical transfer: connecting the new material to be learned with the knowledge that already exists in memory [Hoc 1983, Papert 1980, Bonar and Soloway 1985, Anderson 1985, Bayman and Mayer 1984]. When presenting a novice with a description of a problem during the guided discovery programming phase, a discovery programming system should also allow him to look at several example cases or solutions to different but similar problems. These example cases should be designed to have a close mapping onto the current problem. The novice should be able to use the example solution as a model for his own solution by transforming the whole or a part of the example solution into his own solution, replacing and modifying only those individual elements of the

example solution that do not satisfy the new requirements.

2.3 Intelligent Coaching

A discovery programming system develops the novice's programming skill by providing him with a guided discovery programming phase. In this phase, the novice composes and relates different programming concepts and language constructs to form complete algorithms for given problems. The system monitors the novice's actions as he moves along the solution path, automatically analyzes partial solutions for semantic errors and misconceptions, and offers intelligent [illegible] partial

Many of the automatic program debugging systems, including Proust [Johnson 1984], Talus [Murray 1986], Aurac [Hasemer 1983], Bip [Barr 1987], and Luara [Adam and Laurent 1980], cannot debug partial code segments and wait until the entire program code is completed before attempting any debugging analysis. As a result, these systems lack any rich interaction with novices during program construction and require them to possess a high level of both programming knowledge and programming skill. Automatic program debuggers embodied and incorporated in a discovery programming system should be capable of debugging partial solutions as they are provided by novices. This feature is mandatory for a discovery system to be able to monitor novices' progress in putting programming concepts and language constructs together, and decide when to interrupt and what to say. Novices can only be expected to have partial programming knowledge of how programming concepts and language constructs work, how they affect the underlying notional machine and how the machine executes and treats whole programs. In a discovery system, this knowledge is expected to be gained during the free discovery programming phase. It is the task of the discovery programming system, through its guided discovery programming phase, to help novices transform their programming knowledge into programming skill.

Several programming systems and tutors support the debugging of partial solutions, among which are the Lisp Tutor [Anderson 1985], Bridge [Bonar 1988], and Gil [Reiser 1988]. Unlike Bridge, which requires the novice to specifically request the automatic analysis of his program (passive-like mode), a discovery programming system, like the Lisp Tutor and Gil, should support active, automatic debugging for it to be able to monitor each and every step that novices may take while moving on a solution path, determine when novices show evidence of misconceptions and decide when to interrupt and what to say. This requires support for possible immediate feedback on both failure and success. However, immediate feedback should not spoil the spirit of discovery learning, and should not impose on novices the rigidity found in the early version of Lisp Tutor, for example. This can be achieved by supporting a more flexible style of tutorial interaction that:

1. Increases the grain size of automatic debugging to a complete expression or statement, not just a single symbol,

2. Permits the user to enter the code in any order, not just left-to-right, and

3. Allows the user to backtrack and delete previously entered code.

By supporting these features, a discovery programming system combines the virtues of both discovery learning and immediate feedback. A further discussion on these principle design issues can be found in Ramadhan [1991].

A new implementation of the Lisp Tutor, called a student-controlled tutor [Anderson 1990], attempts to ease the rigidity found in the classic version of the tutor. In addition to providing novices with the three features mentioned above, the new tutor also enables them to control the timing of the feedback. However, these features are supported only in the program editing mode during which the tutor has no interaction with the novice. This transition from tutor-controlled interaction to student-controlled interaction makes the new tutor, like Bridge, a passive system that waits for the novice to request automatic analysis of his code, and thus loses the rich interaction with him. In addition, when the novice asks for automatic debugging of his code, the tutor checks over the code in the same sequence as the original version: top-down, left-to-right. Feedback is given on the first error found and the rest of the code is just ignored. Although the new implementation is an improvement, it takes away a very important feature from the tutor: the ability to monitor the novice's progress on the solution path, determine when he shows evidence of misconceptions within their proper and immediate context and decide when to guide him and what to say during interactive tutoring.

3 Conclusion

This paper discusses a framework for designing and developing a discovery programming system. It is argued that a truly robust discovery programming system must be able to allow novices acquire both programming knowledge and programming skill. A discovery system supports novices in the initial free discovery programming phase and the subsequent guided discovery programming phase. In the initial phase, novices observe and discover the dynamic behavior of individual programming concepts as well as of whole programs to build the underlying conceptual programming knowledge. In the subsequent phase, novices compose and coordinate programming concepts and language constructs, observed and discovered in the initial phase, together to solve given problems under explicit intelligent guidance of system's domain expert in order to transform their programming knowledge into programming skill.

The integration of visibility and program visualization within a concrete model of the underlying notional machine, coupled with the case-based reasoning

and the immediacy of intelligent tutoring are expected to provide a discovery programming system with a potential to combine virtues and features of both HCI and ITS to teach novices basic computer programming in a dynamic and conceptually rich way.

Acknowledgement

I am deeply greatful to Ben du Boulay for many helpful discussions, suggestions and stimulating intellectual challenges to do with my research, and also for critically reading earlier versions of this paper. This research is a part of an ongoing PhD study supported by a full scholarship grant from the Sultan Qaboos University, Oman.

References

[Adam and Laurent 1980] Adam, A. and Laurent, J. LAURA, A System to Debug Student Programs. *Artificial Intelligence* 15 (15): 75-122, 1980.

[Anderson 1982] Anderson, J. R. Acquisition of Cognitive Skill. *Psychological Review* 89, 369-406.

[Anderson 1985] Anderson, J. R., Boyle, C. F. and Reiser, B. J. Intelligent Tutoring Systems. *Science* 228, 456-462.

[Anderson 1990] Anderson, J., Boyle, C., Corbet, A. and Lewis, M. Cognitive Modeling and Intelligent Tutoring. *Journal of Artificial Intelligence* 42, 7-49.

[Barr 1976] Barr, A., Beard, M., and Atkinson, R. The Computer as a Tutorial Laboratory. *International Journal of Man-Machine Studies* 8, 567-596.

[Bayman and Mayer 1984] Bayman, P. and Mayer, R. Instructional manipulation of user's mental models for electronic calculators. *International Journal of Man-Machine Studies*, 20, 189-199.

[Bonar 1985] Bonar, J. G. *Personal Programming in BASIC.* Academic Press, USA.

[Bonar 1988] Bonar, J. G. Intelligent Tutoring with Intermeidiate Representations. *Proceedings of the First Conference on Intelligent Tutoring Systems, ITS-88.* Montreal, Canada.

[Bonar and Soloway 1985] Bonar, J. and Soloway, E. Preprogramming knowledge: a major source of misconceptions in novice programmers. *Human-Computer Studies in Mathematics*, 20, 293-316.

[Brown 1986] Brown, M. *Algorithm Animations*. Ph.D. thesis, Brown University, USA, 1986.

[Chi 1982] Chi, M. T. H., Glaser, R. and Rees, E. *Expertise in Problem Solving*. In Strenberg, R. (editor), Advances in the Psychology of Human Intelligence Lawrence Erlbaum and Associates, Hillsdale, New Jersey.

[Dijkstra 1982] Dijkstra, E. W. How do we Tell Truths that Might Hurt? *SIGPLAN Notices*. 17(5): 13-15. May.

[du Boulay 1981] du Boulay, J.B.H., O'Shea, T. and Monk, J. The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices. *International Journal of Man-Machine Studies*, 14, 237-249.

[du Boulay 1986] du Boulay, J.B.H. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*, 2, 57-63.

[du Boulay 1987] du Boulay, J.B.H., Taylor, J. *Computers, Cognition and Development*. J. Wiley and Sons, 1987.

[Eisenstadt 1991] Eisenstadt, M., Rajan, T. and Keane, M. *Novice Programming Environments*. Ablex Publishing, Brighton, UK.

[Green 1990] Green, T. Programming Languages as Information Structures. In Hoc, Green, Samurcay and Gilmore (Eds.), *Psychology of Programming*, Academic Press, London.

[Hasemer 1983] Hasemer, T. *An Empirically-Based Debugging System for Novice Programmers*. Ph.D. thesis, The Open University, UK, 1983.

[Hoc 1983] Hoc, J. Analysis of beginner's problem solving strategies in programming. In Green, Payne and Van der Veer (Eds.), *The Psychology of Computer Use*, London: Academic Press.

[Johnson 1985] Johnson, W. *Intention-Based Diagnosis of Errors in Novice Programmers.* Ph.D. thesis, Yale University, USA, 1985.

[Jones 1984] Jones, A. How Novices Learn to Program. *Proceedings of the First IFIP Conference on Human Computer-Interaction*, INERACT-84, London, UK.

[Lieberman 1985] Lieberman, H. Seeing what your programs are doing. *International Journal of Man-Machine Studies*, 19, 253-271.

[Mayer 1981] Mayer, R. E. The Psychology of How Novices Learn Computer Programming. *Computing Surveys* 13, 121-141.

[Miller 1982] Miller, J., Kehler, T., Michaels, P. and Murray, W. *Intelligent Tutoring for Programming Tasks.* Technical Report, Texas Instruments, 1982.

[Moran 1982] Moran, T. and Card, S. Applying cognitive psychology to computer systems: A graduate seminar. In Moran, T. (Ed.), *Eight Short Papers in User Psychology*, Palo Alto, CA, USA: Xerox.

[Murray 1986] Murray, W. *Automatic Program Debugging for Intelligent Tutoring Systems.* Ph.D. thesis, Texas University, Austin, USA, 1986.

[Myers 1988] Myers, B. A. The State of the Art in Visual Programming and Program Visualization. *Carnegie Mellon University Technical Report*, Computer Science Department, Carnegie Mellon University, USA.

[Nievergelt 1978] Nievergelt, J. XS0: A Self Explanatory School Computer. *SIGCE Bull* 10, 66-69

[Norman 1983] Norman, D. Some Observations on Mental Models. In Gentner and Stevens (Eds.), *Mental Models*, Hillsdale, NJ, USA: Erlbaum.

[Olson 1986] Olson, G. and Gugerty, L. Comprehension differences in debugging by skilled and novice programmers. In Soloway and Iyengar (Eds.), *Empirical Studies of Programmers*, Norwood, NJ, USA: Ablex.

[Papert 1980] Papert, S. *Mindstorms: children, computers and powerful ideas*. New York: Basic Books.

[Pea 1986] Pea, R. D. Language-Independent Conceptual 'Bugs' in Novice Programming. *Journal of Educational Computing Reseaech* 2. 25-36.

[Rajan 1991] Rajan, T. *Novice Programming Environments*. Ablex Publishing, Brighton, UK.

[Ramadhan 1991] Ramadhan, H. A Discovery Programming System. *Cognitive Science Research Papers*. Sussex University, Brighton, UK.

[Reiser 1988] Reiser, B., Kimberg, D., Lovett, M. and Ranney, M. Knowledge Representation and Explanation in GIL, an Intelligent Tutor for Programming. *Cognitive Science Laboratory Reprot*. Princeton University, NJ, USA

[Schweppe 1973] Schweppe, E. J. Dynamic Instructional Models of Computer Organization and Programming Languages. *SIGCE Bull* 5, 26-31.

[Shapiro 1974] Shapiro, S. C., and Witner, D. P. Interactive Visual Simulations for Beginning Programming Students. *SIGCE Bull* 6, 11-14.

[Soloway 1984] Soloway, E. and Ehrlich, K. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering. Special Issue: Reusability*, Sept.