Writing, Reading, Watching: A Task-Based Analysis and Review of Learners' Programming Environments

Tim Wright and Andy Cockburn
Department of Computer Science
University of Canterbury, Christchurch
{*tnw13,andy*}*@cosc.canterbury.ac.nz*

Abstract

This paper identifies three fundamental learning activities in the development of literary skills—writing, reading, and watching—and describes the potential benefits of supporting these activities when learning to program computers. We analyse the support for writing, reading and watching provided by current educational programming environments and show that no current systems offer comprehensive and integrated support for the three activities. In particular, support for watching the relationship between the program code and the resultant program behaviour is poor.

1 Introduction

The ability to use computers is rapidly becoming an essential skill for everyday life. The ability to *program* computers, however, remains largely in the hands of highly trained professionals. The disparity between normal use and programming is changing, and the rate of change is likely to increase as users' needs and capabilities demand more from their computing environments. Already many home appliances, such as microwave ovens and VCRs, require elements of programming skills to operate them effectively, and most 'end-user' computing applications enable much higher levels of efficiency if users know a few fundamentals of programming: examples include spread-sheet applications, mail-merge facilities and macro recorders.

Since Papert invented Logo in the 1970s [9] there has been substantial interest in programming environments for children. Educationalists have attempted to understand the merits of teaching programming as a tool for promoting 'structured thinking' and 'constructivist learning' [11] (for example, [6]). Computer scientists have developed *many* diverse systems that attempt to improve children's ability to program. The focus of many of these systems has been on developing innovative programming paradigms that have ranged from graphical rewrite rules through to video-game metaphors. Despite these fascinating technical developments, there has been a lack of investigation into fundamental learning activities in programming.

In this paper, we provide a 'first principles' analysis of three fundamental learning activities—writing, reading and watching—that should be supported by all learners' programming environments. We argue that these activities offer a natural scaffolding structure that promotes learning and can encourage transfer effects that ease the transition from elementary to advanced programming concepts. We also show, by review, that these tasks are not supported in an integrated manner by current state-of-the-art educational programming environments. This analysis provides design guidance that we will use in continuing our development of learners' programming environments.

2 Writing, Reading, Watching

Vygotsky [20] argues that interaction with adults and more competent peers is a pivotal factor in effective learning. Rose's studies of children learning how to read and write amplify this issue: "one of the best predictors of reading and writing success is the amount of 'expert' reading children have seen in the home as their parents read to them or write stories as children dictate" [14]. We use the term 'watching' to encapsulate this learning process of observing actions and interacting with more competent peers.

When learning how to program computers there are two sources of more competent peers that the student can observe and interact with to test and build their understanding. First, the student can work with human collaborators (teachers or fellow students). This may be through normal face-to-face interaction (such as that supported by AlgoBlock [18], see Section 3), through some form of groupware support (for example, Cleogo [3], see Section 3.4), or through a computerised agent that emulates human behaviour.

Second, and the focus of this paper, the computer

itself provides an interactive platform for experimentation. When the student writes a program, its behaviour is a dynamic embodiment (the watchable form) of the program (the readable form) that the student has expressed (the written form). Using natural language as an analogy, this is equivalent to the student expressing a verbal phrase (for instance, saying the words "The cow runs"), then seeing *both* the words `The cow runs` and an animation of a running cow on the screen (Figure 1).

We use the terms writing, reading and watching to encapsulate three fundamental activities in working with computer programs. *Writing* is the process of recording or encoding a series of program actions. The writing process may be supported by a standard text based language (such as C), through visual symbols (for example, programming graphical user interfaces in Visual Basic or HyperCard), or through a programming by example interface such as Eager [5]. *Reading* is the process of reviewing the static representation of a previously recorded program. *Watching* is the process of mapping between the learner's understanding of the program and the observed dynamic behaviour of the running program.

When programming, users manipulate various symbols in order to specify the program's behaviour within the output domain. These symbols can vary from Fregan [17], abstract, representations of the concepts they encode (for example, `weight++`) through iconic representations (an iconic depiction of a person's swelling belly) to tangible objects (an inflatable robot). Additionally, these symbols can exist at many different levels of abstraction from the problem domain.

2.1 Programming Gulfs

By separating the three activities of writing, reading and watching, and by giving a user-centred perspective to the symbols used for each, we aim to illuminate the learning problems that users have when mapping between their understanding and the mechanisms provided by the programming environment. Extending Norman's user-centred design concepts of the gulfs of execution and evaluation [8], we us the terms 'gulf of expression', 'gulf of representation' and 'gulf of visualisation' to describe these mapping difficulties (see Figure 1).

Gulf of Expression. Programmers must translate their internal mental model of the desired program behaviour into the syntax and symbols of the programming language. This gulf causes a trade-off in language design between the level of abstraction of the language's symbols and its potential generality. Conventional programming languages, such as C or Java, use abstract symbols for program expression, causing a large gulf between the language of expression (textual symbols) and

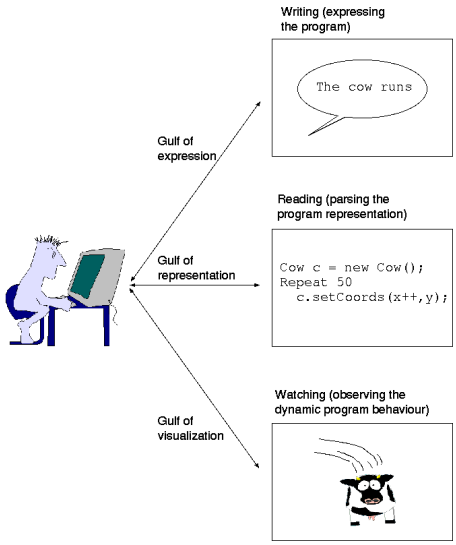

Figure 1: Gulfs of expression, representation and visualisation.

the resultant program behaviour (possibly an animation or a graphical user interface), but the abstract symbols enable the language to program a wide range of different domains. In contrast, languages with a strong mapping between their symbols and the domain will normally be constrained to a small set of domain-specific problems. For example, programming by demonstration systems, in which the input symbols are identical to the programming domain, are normally limited to a specific domain (see [10] for a discussion of domain independent programming by demonstration).

Another factor influencing the gulf of expression is the extent to which the programming environment constrains (or 'determines') the types of expressions that the programmer can issue. Conventional programming languages typically under-determine the user by providing no constraints on the textual symbols that the user can enter. However, research systems, such as the Cornell Program Synthesizer [19], over-determine the programmer by requiring that statements be selected through the language's grammatical rules: the resultant programs are guaranteed to be syntactically correct, but the user is forced to work through excessive constraints. More recently, many commercial programming environments include type-ahead facilities that allow programmers to select from contextual information when available (such as the methods of an object of a particular class), while also allowing any symbols to be typed.

Gulf of Representation. The readable form of the program provides a surrogate for the programmer's mental model. In order to understand a program, the

programmer must read the code (or any associated documentation) to construct their mental model of its behaviour. The cognitive mapping between the readable program form and the programmer's mental model causes a gulf of representation.

There is a relationship between the gulf of expression and the gulf of representation. Programs are expressed at a certain level of abstraction, and using particular symbols. If they are not represented for reading in the same way, then the programmer must master multiple symbol sets (see Figure 1).

The gulf of representation is exacerbated by the difficulty of browsing program code. Programs consist of interconnected procedures, rules, methods, etc., and the user must navigate through this rich hypertextual space to comprehend the program. Rader [12], for instance, reported that StageCast's lack of a visualisation component caused difficulties for children in mapping between the agent-based representation of rules and their mental model of expected behaviour (see Section 3.2).

Gulf of Visualisation. A gulf of visualisation arises when a programmer has difficulty mapping between the observed behaviour of the running program and their mental model. It is important to note that the program representation serves as a surrogate for the user's mental model of the program. Programmers, therefore, must map between their program's observed behaviour and their mental model as encoded in the program representation. The gulf of visualisation is not only a problem for novice programmers; it also causes problems for competent programmers who must use debuggers and other tools to overcome the difficulties of visualising the internal dynamic behaviour of the program.

Within educational environments the gulf of visualisation provides opportunities for scaffolding the student's understanding. Systems could allow learners to manipulate the visualised behaviour of the environment in a variety of ways, including changing and controlling the timing of execution of statements (for instance, stepping forwards and backwards through a series of instructions), and revealing internal structures that are not normally viewable (for instance, the state of the runtime stack). By providing controllable insights into the machine's state (through appropriate metaphors), we believe that the programming environment can encourage transfer effects between novice and more advanced programming concepts.

2.2 Reducing the gulfs

To re-cap, the gulf of expression refers to the cognitive difference between the user's mental model of the problem domain and the mechanisms used to write the program. The gulf of representation refers to the difference between the user's mental model and the readable program. The gulf of visualisation refers to the difference between the user's mental model of the program (normally supported by a readable surrogate) and the behaviour of the program.

The gulfs of expression and representation can be reduced either by educating the users so that their mental model moves closer to the computational model (the conventional approach to programming), or by bringing the programming mechanisms closer to the user's model [16]. They can also be reduced by using the same symbols for expression and representation: the user need only learn one set of mappings between the programming domain and the symbols used to manipulate it. The gulf of visualisation can be reduced by improving the mapping between three elements: the visual display of the program's behaviour, the visibility of the state of the machine, and the representation of the program statements being executed. Finally, we hypothesise that by appropriately designing the visualisation capabilities of educational programming environments, it is possible to enhance understanding of programming concepts.

3 Current Systems

Language	Symbol Type: Fregan	Mixed	Visual	Tangible
Domain centred				
Logo, Alice				
Domain			★✔★	
Writing	✔			
Reading	✔			
Watching				
Electronic Block				
Domain				★✔★
Writing				✔
Reading				✔
Watching				✔
ToonTalk				
Domain			★✔★	
Writing			✔	
Reading				
Watching			✔	
Writing centred				
Algoblock				
Domain			✔	
Writing				★✔★
Reading				✔
Watching			✔	✔
StageCast, Agentsheets				
Domain			★✔★	
Writing		✔	★✔★	
Reading		✔	✔	
Watching			✔	
Reading centred				
Leogo				
Domain			✔	
Writing	✔	✔	✔	
Reading	★✔★	★✔★	★✔★	
Watching				
Watching centred				
MacBalsa				
Domain				
Writing				
Reading	✔			
Watching			★✔★	

Table 1: Classification of languages: domain, writing, reading and watching centred. ★✔★ indicates the design focus.

This section reviews the support for writing, reading and watching provided by educational programming systems. The review is structured by grouping systems

that focus on the domain that they support, and those that focus on support for writing, reading and watching programs. Table 1 provides a summary comparison of the symbols used for the domain, writing, reading and watching: whether they are Fregan (e.g. 'forward'), mixed (e.g. a button depicting a forward arrow), visual (e.g. dragging a turtle forward with the mouse), or tangible (e.g. dragging a turtle robot forward).

3.1 Domain centred systems

Many educational programming systems focus on providing the learner with motivating, engaging and familiar programming domains. For example, Papert's original version of Logo [9] and recent systems such as Alice [4] are designed around their graphical output domains: 2D turtle graphics in Logo, and a 3D object-oriented world in Alice. Both of these systems provide Fregan, text-based, symbol sets for reading and writing programs ('forward', 'left', 'alice.head.turn(1)', etc.) Fregan languages, however, cause gulfs of representation and expression for youthful users who may have difficulty comprehending the syntax and symbols, as well as problems with keyboard use. Logo and Alice provide minimal support for watching programs run: the complete program (or procedure call) is run in a single step, and there is no way for the programmer to watch the relationship between each instruction and the system state.

ToonTalk [7] is programmed by issuing instructions by demonstration to graphical robots within an environment that is based on a video-game metaphor. Its gulf of expression is low—the robot learns the demonstrated actions—but its gulf of representation is large because there is no support for reviewing the program associated with each robot. Future work on ToonTalk will ease this gulf by providing a 'time travel' metaphor that will allow the programmer to revisit the actions shown to a robot. The absence of support for reading leads to a gulf of visualisation because there is no readable surrogate for the user's mental model of the program.

Electronic Blocks [21] is a tangible language for children. Children write programs by connecting electronic Lego-like blocks together, where each block represents a logic statement (`and`, `not`), a delay, an input (`light`, `noise`, `touch`), or an output (`light`, `noise`, `movement`). Although a fascinating concept, the paper describing Electronic Blocks was published prior to implementing or evaluating the language, and it is unclear how the gulfs of representation, expression and visualisation will affect the user's interaction.

3.2 Writing centred systems

Several educational systems have investigated improved ways of allowing children to write programs. In AlgoBlock [18], for instance, programs are written by connecting physical blocks in a similar manner to Electronic Blocks. Each block represents a LOGO primitive, and symbols on the block are used to depict its action. The output of AlgoBlock programs is displayed on a computer screen, creating a gulf of visualisation between the program representation (the blocks) and the output domain. AlgoBlock slightly eases this gulf by illuminating a light on each block as its associated statement is executed.

StageCast [15] and Agentsheets [13] are designed to let users program graphical simulations. They minimise the gulf of expression by using graphical rewrite rules based on 'before and after' conditions. The gulf of representation is reduced by showing rules associated with each graphical 'agent' in the same form as they were entered. When the program runs, a light on each rule illuminates whenever it fires, helping to reduce the gulf of visualisation. Furthermore, the user can change the speed at which rules are fired, allowing the user to step-through a series of rule-firing actions. The main limitation in the support for watching is that only one agent's rules can be displayed at a time, therefore providing a constrained view of the overall program behaviour [12]. AgentSheets adds a powerful 'analogous examples' capability to ease writing programs: for example, having programmed a car agent to follow a road, a train agent can be programmed to follow a track by stating that it does so *like* a car follows the road.

3.3 Reading centred systems

Leogo [2] was designed to test the concept of 'equal opportunity programming' which aims to reduce the boundaries between a programming language and its output domain. Leogo provides three parallel 'programming paradigms' for writing Logo programs: a Fregan text-based dialect of Logo; an iconic representation of Logo statements and procedures; and a set of direct manipulation turtle manipulations. Actions expressed in any of these three schemes causes immediate output of an equivalent statement in the other two paradigms. The educational objective was to encourage transfer effects between the three paradigms by allowing the user to read equivalent statements in each language.

Other than showing the translated statements in parallel, Leogo offers no support for watching the relationship between the program representation and its behaviour.

3.4 Watching centred systems

Although several systems have provided some form of support for watching programs execute (see Table 1), there has been a surprising lack of work explicitly focusing on this fundamental learning activity. The notable

exception is MacBalsa [1] which allowed programming instructors to specify animations that would be used to clarify algorithm execution in lectures to tertiary-level students. Although MacBalsa allowed students to read the program code and to watch its execution, it did not provide support for writing programs.

Cleogo [3], a synchronous groupware version of Leogo (Section 3.3), and AlgoBlock support watching through their support for collaboration around the programming environment. This is a different interpretation of 'watching' than the one used elsewhere in the paper, but one that is no less important. Rather than learning programming behaviour by watching and comprehending the relationship between the system behaviour and the program code, the users collaborate in their expression of the program, and have an opportunity to articulate and mutually reinforce each others' understanding.

4 Summary and Further Work

We have argued that watching is a fundamental learning activity that is poorly supported in current educational programming environments. The current focus of programming environments for learners is on motivating and engaging students, and on providing *easier*, more natural, paradigms for programming. While applauding these developments, we believe that improved support for watching program execution will help to appropriately scaffold the student's learning. In our future work we will develop and evaluate programming environments that explicitly support an integrated approach to writing and reading programs, and we will particularly focus on support for watching the relationship between the represented program and its behaviour.

Acknowledgements

Thanks to Lauri Ricker for her useful comments. This work is supported by a New Zealand Marsden grant.

References

[1] MH Brown and R Sedgewick. Techniques for algorithm animation. *IEEE Computer*, 2(1):28–39, 1985.

[2] A Cockburn and A Bryant. Leogo: An equal opportunity user interface for programming. *Journal of Visual Languages and Computing*, 8(5-6):601–619, 1997.

[3] A Cockburn and A Bryant. Cleogo: Collaborative and multi-paradigm programming for kids. In *APCHI'98: Asia Pacific Conference on Computer Human Interaction. Japan. July 15–17*, pages 187–192. IEEE Computer Society Press, 1998.

[4] Matthew J. Conway. *Alice: Easy-to-Learn 3D Scripting for Novices*. PhD thesis, University of Virginia, 1997.

[5] A Cypher. Eager: Programming repetitive tasks by example. In *Proceedings of CHI'91 Conference on Human Factors in Computing Systems* New Orleans, May, pages 33–39, 1991.

[6] AL Fay and RE Mayer. Benefits of teaching design skills before teaching logo computer programming: Evidence for syntax-independent learning. *Journal of Educational Computing Research*, 11(3):187–210, 1994.

[7] K Kahn. ToonTalk—an animated programming environment for children. *Journal of Visual Languages and Computing*, 7(2):197–217, 1996.

[8] DA Norman. *The Psychology of Everyday Things*. London: Basic Books, 1988.

[9] S Papert. *Mindstorms — Children, Computers, and Powerful Ideas*. Harvester Press, Brighton, 1980.

[10] G Paynter. *Domain Independent Programming By Demonstration*. PhD thesis, Department of Computer Science, University of Waikato, New Zealand, 2000.

[11] J Piaget. *To Understand is to Invent: The Future of Education*. Grossman, 1973.

[12] C Rader, C Brand, and C Lewis. Degrees of Comprehension: Children's Understanding of a Visual Programming Environment. In *Proceedings of the ACM SIGCHI'97 Conference on Human Factors in Computing Systems,* Atlanta, Georgia, March 22-27, pages 351–358, 1997.

[13] A Repenning and C Perrone. Programming by Analogous Examples. *Communications of the ACM*, 43(3):90–97, 2000.

[14] D Rose. Apprenticeship and exploration: A new approach to literacy instruction. In *CAST Literacy research papers (www.cast.org)*, number 6. New York: Scholastic, 1995.

[15] DC Smith, A Cypher, and J Spohrer. KidSim: Programming Agents Without a Programming Language. *Communications of the ACM*, 37(7):55–67, 1994.

[16] DC Smith, A Cypher, and L Tesler. Novice programming comes of age. *Communications of the ACM*, 43(3):75–81, 2000.

[17] A Soloman. Interactions between philosophy and artificial intelligence: The role of intuition and non-logical reasoning in intelligence. In *Proceedings of the 2nd International Joint Conference on Artificial Intelligence (London)*, pages 270–278. Morgan Kauffman, San Francisco, 1971.

[18] H Suzuki and H Kato. Interaction-level support for collaborative learning: *AlgoBlock* — an open programming language. In *ACM Conference on Computer Supported Cooperative Learning (CSCL '95). Bloomington, Indiana. October 17–20*, pages 349–355. Lawrence Erlbaum Associates, Inc, 1995.

[19] T Teitelbaum. The cornell program synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, 1981.

[20] LS Vygotsky. *Mind in Society*. Harvard University Press, 1978.

[21] P Wyeth and HC Purchase. Programming Without a Computer: A New Interface For Children Under Eight. In *Proceedings of the Australian User Interface Conference,* Canberra, Australia, 31 Jan–3 Feb, pages 141–148, 2000.