Recognizing Opportunities for Mixed-Initiative Interactions in Novice Programming

Liam Doherty, Patrick Lougheed, David Brokenshire, Mayo Jordanov, Shilpi Rao, Jurika Shakya, and Vive Kumar

Simon Fraser University, Canada
ldoherty@sfu.ca, vive@sfu.ca

Abstract

As part of their education, Computer Science students develop hundreds of computer programs and receive only summative feedback on the end results of their program designs and code. We demonstrate a need for a formative feedback mixed-initiative system able to reflect on the progress of students and identify opportunities to prompt students when they venture into 'poor' programming styles. With the eventual goal of developing a mixed-initiative system with these capabilities, able to make use of analysis of compile-time code segments, theories from research into self-regulated learning are applied to the problem of identifying opportunities for mixed-initiative interactions.

Objectives, Purpose and Theoretical Framework

The instructional design of many introductory programming courses in computer science does not include introduction to 'good' programming styles. In most cases, these instructional designs provide only summative feedback to students on their output, ignoring formative feedback as students design and code their programs (Spohrer and Soloway 1986; Corbett and Anderson 1992). As a result, students may develop improper or undesirable styles of coding which may make future refinement and amalgamation with the coding of others difficult. However, poor coding styles may be remedied by capturing the process of how novice students learn to program and by providing system-initiated process (formative) feedback about their programming behaviour. Process feedback is information about how students engage in a task (Early et al. 1990). In this paper, we consider means to develop a mixed-initiative system to proactively prompt and provide feedback on students' styles in designing and constructing their computer programs. This system makes use of data from a tool for analysis of compile-time code segments.

One of the problems faced by novice programmers is the leap from an understanding of individual language constructs to an understanding of groups of constructs. Liffick and Aiken (1996) present a model that represents programming knowledge in five distinct cognitive levels, *lexical*, *syntactic*, *semantic*, *schematic*, and *conceptual*, to annotate novice programs and address the 'leap' problem. Analysis of compile-time code segments can be used to examine all five cognitive levels. For example, the lexical and syntactic changes in the code across various compilations can be captured and students who made a particular set of lexical and syntax changes can receive corresponding feedback. The semantic and schematic level changes correspond to individual and multiple statements respectively. Such changes across multiple compiles can be manually or semi-automatically captured and the types of such changes can be indexed. The schematic level changes reflect and track how exactly student evolve their programming plans.

The mental model theory (Gentner and Stevens 1983; Borgman 1986; Payne 1988) advocates that mental models can be enriched when appropriate conceptual models are identified and used to enhance the states of entities and their interrelationships (Yu-Fen and Alessi, 1993). Compile-time code analysis can be used to 'recognize' conceptual models of programming practices of students, in terms of language constructs and compiler output. This data can be used by a mixed-initiative system to help students develop better mental models of their programming styles. For example, students can be instructed to develop code based on the conceptual model that advocates a linear step-wise increase in the number of lines of code. Students' code compiles can be verified against this model to find how well their coding processes correspond to the model.

Madeo and Bird (Madeo and Bird 1990) found that, to be effective, weaker students should be taught programming in a very tightly structured environment. The proposed mixed-initiative interface presents a context that not only customizes the environment to the specific needs of the student but also helps the teacher to individualize their instruction to the needs of the students.

Compile-Time Analysis

A computer program consists of language constructs. Computer programming is an iterative process which cycles through conceptualization, construction, verification, and revision of constructs. In general, the nodes of the Abstract Syntax Tree (AST) of a programming language form the constructs of the language

at the leaf nodes or at higher abstraction levels. For example, the code in Figure 1 is comprised of many 'C' language constructs.

```
[
 int temp;
 char s[200],new[200];
 n_start = 0;
 gets(s);

 while )temp >= 0)
```

Figure 1: Code segment – compile version 1

When novice students learn to program, they conceptualize the program design to tackle the problem at hand, build code using the constructs, verify the constructs using manual (code review) or automated (compiler) tools for correctness, and revise the constructs based on the feedback from the tools and other cognitive and/or programmatic considerations. These programming activities are situated within the context of a programming language, an integrated development environment (IDE), a compiler, an operating system, and a hardware platform.

In a typical programming exercise, students start to write code and from time to time submit their code to the compiler for verification. When a student submits his/her partially or fully developed code to a compiler, the code that goes through the verification process is identified as a compile-time code segment (CT-SEG). The compiler returns a list of errors with respect to the constructs corresponding to the code segment submitted for compilation.

```
#include <stdio.h>
main () [
 int temp;
 char s[200],new[200];

 gets(s);
 while (temp >= 0)
```

Figure 2: Code segment – compile version 2

Most novice students write code that is syntactically and semantically erratic with respect to the constructs, and they tend to use the feedback from the compiler to revise their code. They submit the revised code to the compiler again for verification. This process continues until students successfully complete their code that is syntactically and semantically correct or relinquish/postpone the process due to other constraints. For example, Figure 2 is the next version (CT-SEG) of code that the student submitted for compilation with minor changes to the constructs.

Novice students tend to generate many CT-SEG corresponding to the many versions of code that he/she submits for compilation. For example, the study that we ran (Kumar 2004) showed that most novice students compiled their code (in 'C' language) 16 times in a 90-minute time-limited exercise, on a Unix operating system, using a command line gcc compiler on a PC.

Method and Data Sources

The study was set up as part of a lab assignment in an introductory 'C' programming language course. The students were asked to develop an interactive 'C' program to move a character pointer to different parts of a sample sentence. The students were given access to a UNIX operating system, gcc compiler, and vi editor, in a PC platform. The time limit was set as 90 minutes to complete the lab assignment.

96 students participated in the study. Every time a student submitted his/her intermediate code for compilation, the code along with its CT-SEG version number and the time of compilation were recorded in a repository. Thus, if a student compiled his/her code 10 times, there would be 10 CT-SEG versions in the repository along with their respective compilation time stamps. At the end of the study, over 2000 CT-SEG versions were collected, which formed the data source for the study.

The analysis for formative evaluation of programming styles consists of 4 parts: first, a summary of CT-SEG compiles, second, the differential in programming constructs between any two consecutive CT-SEG, third, the compiler output between any two consecutive CT-SEG, and fourth, the relationship between the construct differential among consecutive CT-SEG and the corresponding compiler output.

Results

The results reported here are indicators of how this analysis can be used in various contexts. The results neither confirm any of the theoretical frameworks identified earlier nor explain how well students could learn from the feedback.

Summary of CT-SEG Compile-Time Analysis

The first part of the analysis presents information pertaining to the overall performance of a batch of students. Some of the key indicators include

a) Number of compiles
b) Number of lines of code across CT-SEG
c) Number of errors and warnings across compiles
d) Degree of completion of the assignment (percentage of correctness of the final code)

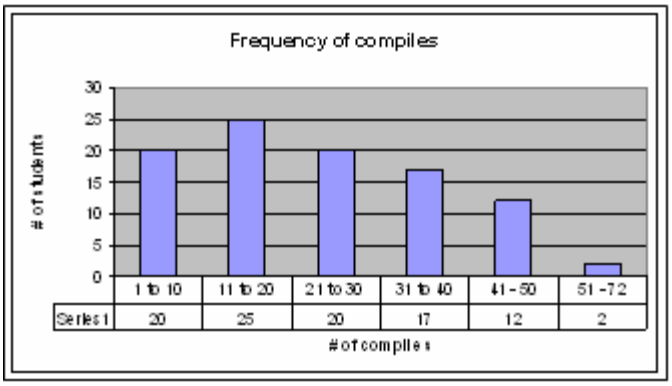

Figure 3: Number of Compiles

In the study we observed the following pattern (in Figure 3) with respect to the number of compiles. Of the 96 students, most of them compiled their code 11 to 20 times. On the extreme side, one student compiled his/her code 72 times in 90 minutes!

Differential in Programming Constructs

The second part of the analysis identifies the differentials in programming constructs between any two consecutive CT-SEG. One such differential is the changes in the number of lines of code (LOC) between CT-SEG. Figure 4 plots the drastic changes in the LOC between compiles for student 1 and Figure 5 plots a smooth increase in the LOC between compiles for student 2.

Similarly, Figure 6 and 7 plot the differential in programming constructs between consecutive CT-SEG for students 1 and 2. Figure 6 indicates that student 1 made drastic changes in 'expression statements' in 6 CT-SEG within 37 compiles. On the other hand, student 2 made considerable changes in 'expression statements' only once in 16 compiles. The analysis can identify such construct-level differentials at the leaf nodes or at any level of abstraction in the AST.

One example of the use of differentials is to teach students the concepts of Dijkstra's structural programming (Dijkstra 1970).

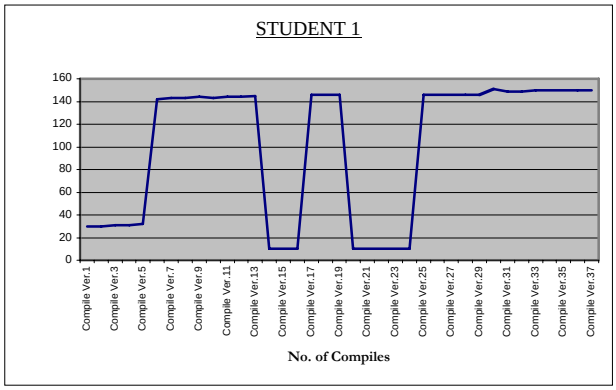

Figure 4: Student 1 Line-of-Code Differential

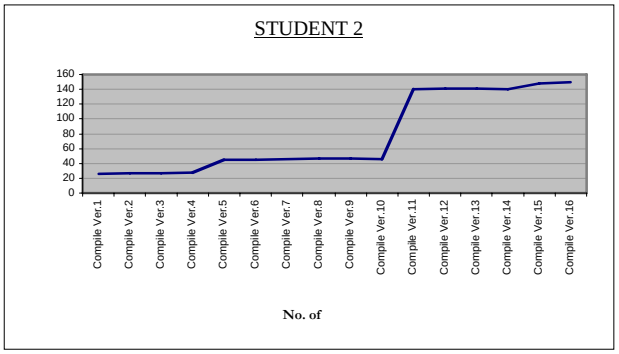

Figure 5: Student 2 Line-of-Code Differential

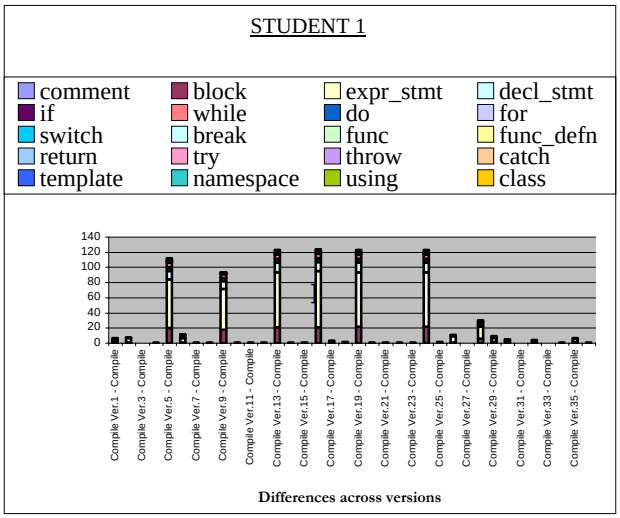

Figure 6: Student 1 Construct Differential

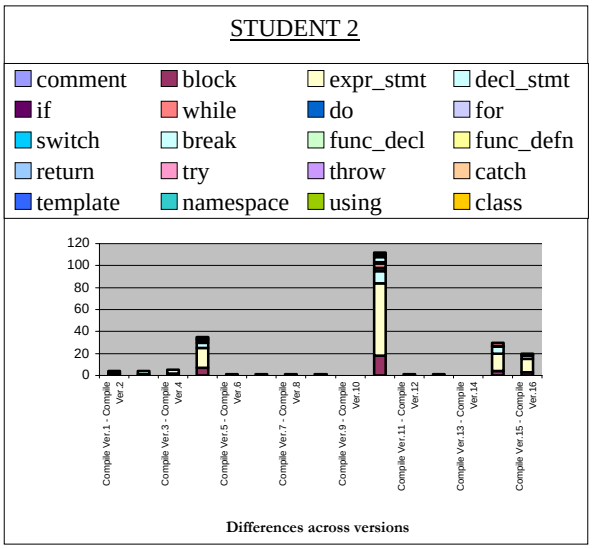

Figure 7: Student 2 Construct Differential

Compiler Output

The third part of the analysis looks at the compiler output between any two consecutive CT-SEGs. Figure 8 plots the errors and the warnings produced by the compiler through 37 compiles of student 1. Figure 9 plots the errors and the warnings produced by the compiler through 16 compiles of student 2. It indicates that student 2 kept ignoring the compiler warnings, which kept growing to a considerable number at 16th compile.

Using compiler output and comparisons between error messages we can find where students often trip up and suggest remedies to avoid the problems in the future. Many of these errors can be proactively fixed by the system, such as missing language syntax. Furthermore, through analysis of errors we may be able to identify when a student needs further help in an area – an opportunity for mixed-interaction.

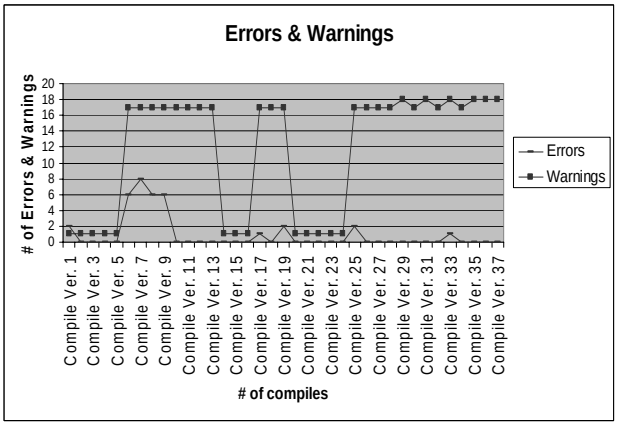

Figure 8: Student 1 Compiler Output

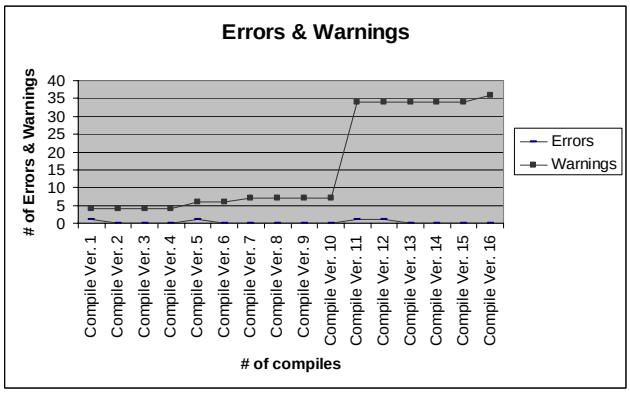

Figure 9: Student 2 Compiler Output

Construct Differential and the Compiler Output

The fourth part of the analysis relates specific errors and warnings in a CT-SEG to one or more construct differentials. For example, a new type of error reported by the compiler in one particular compile can be mapped onto a particular change in code between the corresponding consecutive CT-SEG.

Capturing User Interactions in Ontology

Through ontology we can capture user interactions fully. Interactions that can be captured include those related to the learner browsing course content, searching and browsing the web, modifying code and other interactions inside the IDE, and chatting with instructors or other students. The following is a list of interactions we can capture in our ontology.

Interaction	Description
links	Learner created a link between two sections of course content
highlights	Learner highlighted some course content
browsed	Learner browsed some course content
searchedFor	Learner search for some keywords
startedChatWith	Learner initiated a chat with someone
sendChatMsgTo	Learner sent a chat message to someone
receivedChatMsgFrom	Learner received a chat message from someone
endedChatWith	Learner ended a chat with someone
openedProgrammingProject	Learner opened a project in the programming IDE
closedProgrammingProject	Learner closed a project in the programming IDE
compiled	Learner compiled some code
Ran	Learner ran some compiled class
addedConstruct	Learner added a construct (if statement, for loop, etc.) to a code file
deletedConstruct	Learner removed a construct from a code file

Table 1 - Sample Interactions

This ontology allows us to capture the actions of the user in order to better determine points where the system should take initiative.

Observations	SRL Model Variables	SRL Phase
Internal documentation / note taking	Strategic planning	Forethought phase
Review compiled summary Review / search	Self-reflection	Performance phase
Review compiling patterns	Self-evaluation	Self-reflection phase

Table 2 – Sample mappings of observed behavior to SRL model

Table 1 presents the mapping of learner observations to the SRL model variables and corresponding SRL phases. For instance, the learner interaction recognized as reviewCompiledSummary is mapped onto the SRL variable for self-reflection which belongs to the self-observation in the Performance phase in Zimmerman's SRL model. (Zimmerman 2002)

Applying CT-SEG Analysis Results

The more we know about students' programming knowledge, the better we can teach them. The data provided on the learner's problem-solving strategies and challenges by CT-SEG analysis can be used to inform a mixed-initiative coach as to what to instruct the learner in, and when to intervene to do so.

The process of creating a computer program can be divided into the three phases of self-regulated learning defined by Zimmerman: forethought, performance, and self-reflection. (Zimmerman 2002) These can be seen to correspond roughly to the planning, implementation, and evaluation phases of software development. At the same time, there is one crucial difference: because software engineering involves mapping the learner's mental models into a concrete structure, the learner may stop to reflect, or re-evaluate their planning at any point in the process of implementation – especially when they are learning new concepts through the programming process. Triggering or aiding this process of re-evaluation during the performance/implementation phase (since this is when the mixed-initiative system has access to the user) is a central purpose of the proposed mixed-initiative system.

An important factor observed in the programming process is that novice students paused their programming and compiled their code when they felt the need for verification of the code that they had written so far. Such a pause is an indication of their cognitive state-of-mind and a prominent step in their learning to program – in many cases, this may indicate a point where the user has stopped to evaluate their implementation. We contend that these programming pauses are opportunities for system-initiated interaction. That is, in addition to compiler-supplied feedback, the system can also provide feedback on the programming styles of the learners, guidance based on the underlying SRL model, and scaffold the learners towards best practices in programming. This paper focuses on the identification of opportune moments for system-oriented mixed-initiative intervention, rather than the content of the interactions.

Looking at self-regulated-learning (SRL) theories and the results of research into programming tutors, we can identify some guidelines for when intervention should occur. Our main goal will be to avoid interrupting the user's thought process or 'flow' (Vass, Carroll, & Shaffer, 2002). This means identifying points where the user has either finished a task (such as when the user compiles their program), is unsure of how to proceed, or is making an error so grave that interrupting their thought process is worthwhile. According to cognitive skill acquisition theory, students undertaking programming problems will mainly be in the intermediate stage, where "learners attend to solving problems and learning is focused on how abstract principles are used to solve concrete problems. Here is where flaws in one's knowledge base are corrected" (Chu, 2005). Students who have advanced beyond this to being practiced in applying the subject matter will consequently prefer and benefit from less coaching (Kalyuga, et al., 2003). Customizing SRL rules to take advantage of data from CT-SEG analysis will allow domain-specific challenges to be addressed.

Applying SRL theories to problem-solving in order to generate mixed-initiative interactions presents an interesting challenge, since problem-solving, unlike other learning tasks such as reading, can itself be partly a process of learning strategies for self-regulation. This is especially true of programming, where solving the problem usually involves understanding the problem and the tools available to solve it – in other words, solving programming problems is a learning process. One of the main purposes of the mixed-initiative system should therefore be to encourage self-regulatory abilities of the learner. This means that fadeout of instruction over the period of a course may be appropriate: if the mixed-initiative system continues to coach the user throughout, the user may become dependent on the system to regulate his or her behavior. This is supported by SRL research: Chu suggests that "A guidance-fading effect suggests … [reducing] guidance with increased expertise" (Chu 2002).
In summary, SRL principles suggest that an mixed-initiative system for teaching programming should initiate interactions at the point when the user compiles the program, should aim to encourage self-regulated programming and learning, and should reduce the degree of guidance it provides as the user becomes more experienced. However, our research does not address whether our mapping of SRL principles demonstrated above is correct, only that it is possible to create a computational framework for these interactions.

Future Work

Having developed a partial theoretical basis for the design of a mixed-initiative system for aiding novice programmers which specifies when interaction should occur, the next step is to determine what types of interactions are best suited to assisting learners in this domain (e.g. planning systems, adaptive interfaces, etc.) Finally, we of course need to implement a system and test these theories. The implementation of SRL theories through mixed-initiative is a new and exciting challenge for both fields.

Conclusion

In this paper, a means for converting compile-time code segments into information useful to a formative-feedback mixed-initiative system has been demonstrated. A computational framework for applying self-regulated learning theories to the area of mixed-initiative-supported learning has been, with the conclusion that it is possible to capture the data necessary to evaluate the applicability of self-regulated learning theory.

References

Liffick, B., & Aiken, R. 1996. A novice programmer's support environment. *Proceedings of the SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*. Barcelona, Spain.

Borgman, C.L. 1986. The User's Mental Model of an Information Retrieval System: An Experiment on a Prototype Online Catalog. *International Journal of Man-Machine Studies*, 24(1) :47-64.

Corbett, A.T., and Anderson, J.R. 1992. The Lisp Intelligent Tutoring System: Research in Skill Acquisition. In Larkin, J., and Chabay, R. (Eds.) *Computer assisted instruction and intelligent tutoring systems: Shared goals and complementary approaches*. Hillsdale, NJ: Lawrence Erlbaum.

Gentner, D., and Stevens, A. 1983. *Mental Models*. Hillsdale, New Jersey: Lawrence Erlbaum.

Madeo, L.A., and Bird, D.A. 1990. The effect of user-specified language elements on learning to program. *Research on Computing in Education*, 22(1),:336-348.

Collard, M., Maletic, J.I., and Marcus, A. 2002. Supporting document and data views of source code. In *Proceedings of the 2nd ACM Symposium on Document Engineering*. Virginia, USA: Association of Computing Machinery.

Early, P.C., Northcraft, G.B., Lee, C., and Lituchy, T.R. 1990. Impact of process and outcome feedback on the relation of goal setting to task performance. *Academy of Management Journal*, 33(1): 87-105.

Payne, S.J. 1988. Methods and mental models in theories of cognitive skills. In J.Self (Ed.), 69-87. London, UK: Chapman & Hall.

Yu-Fen, S., and Alessi, S.M. 1993. Mental models and transfer of learning in computer programming. *Research on Computing in Education*, 26(2):154-175.

Spohrer, J.G., and Soloway, E. 1986. Analyzing the high frequency bugs in novice programs, In Soloway, E., and Iyengar, S. (Eds.), 230-251.

Dijkstra, E.W. 1970. Notes on Structural Programming, Technical Report, TH-Report 70-WSK-03, Dept. of Mathematics, Technological University Eindhoven, The Netherlands.

Kumar, V.S. 2004. An instrument for providing formative feedback to novice programmers, In *Proceeding of the Annual Meeting of American Educational Research Association (AERA), Division I – Education in the professions, Paper session –Relationship between teaching and learning (13.032)*, 71, San Diego, CA, USA.

Vass, M., Carroll, J. M., and Shaffer, C. A. 2002. Supporting creativity in problem solving environments. In *Proceedings of the 4th Conference on Creativity & Cognition* (Loughborough, UK, October 13 - 16, 2002). C&C '02. ACM Press, New York, NY, 31-37.

Chu, S.T.L. Bridging the Gap between Research and Practice: Educational Psychology-Based Instructional Design for Developing Online Content. Presented at AERA 2005, Montreal QC, Canada.

Kalyuga, S., Ayres, P., Chandler, P., & Sweller, J. 2003. The expertise reversal effect. Educational Psychologist, 38(1), 23-31.

Zimmerman, B. 2002. Becoming a Self-Regulated Learner: An Overview. *Theory Into Practice. 41(2).* Ohio, USA.