
WHITE PAPER

Continuous Data-driven Learning Assessment

Stephen H. Edwards
Department of Computer Science, Virginia Tech
edwards@cs.vt.edu, +1 (540) 231-5723

Research Question: **How can computing educators evaluate student learning gains as they learn to program?**

VISION AND GOALS

As educators of novice programmers, we are constantly seeking ways to **improve the educational experience of our students**. Regardless of what intervention we employ—whether it is a lab-centric model of instruction, a multiple-choice approach to assessment, or the use of pair programming in the classroom—we lack **tools to evaluate the effectiveness of our interventions** with respect to our students’ programming practices. Our best assessment often comes from grades on examinations, which rarely provide a lens into the day-to-day experiences and learning of our students.

The teaching and study of novice programmers is rife with claims regarding the benefits of learning to program and the power of [insert pedagogic intervention here]. While automatic, on-line protocols are commonplace in human-computer interaction studies, no framework for automatically collecting and analyzing student programming behavior—beginning with code authoring and ending with assessment—exists today. The systematic collection of this data is critical to the teaching of programming from three different perspectives:

- **Researchers** need access to data to conduct and validate experiments across institutional and cultural contexts, as well as provide a common baseline for comparison in new lines of inquiry.
- **Practitioners** need data regarding the behavior and practice of their students to evaluate student learning as well as the effectiveness of new approaches to instruction.
- **Departments and institutions** need data (automatically collected with a minimum of effort) to support outcomes-based assessment at the course and program level.

Indeed, researchers have been taking steps in this direction. In his March 2010 retrospective of the “naughties” (2000–2009), Raymond Lister wrote: “The naughties saw the development of several systems for routinely logging data about computing students (Winters & Payne, 2005; Jadud, 2006; Norris et al., 2008; Edwards et al., 2009) and I am optimistic about CSEd research in a future where we routinely collect such data.” [15]

However, existing research typically looks at the novice programmer through a single lens, whether it be their interactions with an interactive development environment (IDE) [11][6], their unit tests and automatic assessment results [3][4], their performance on examinations [16], or other behaviors that we hope, as researchers, might shed light on how to aid students in learning to program [23]. Data that captures student work from the first steps of development through final submission provides a behavioral foundation for (replicable) studies that explore fundamental questions whose answers will immediately impact student learning: for example, how educators can help students be more successful in learning to write programs.

The over-arching goal of any project aimed at this research question would be to **aid the instructor in developing interventions to support student learning** by providing tools to **evaluate the effect of those interventions**. Such an overarching goal could be achieved by bringing together and refining a number of existing, productive lines of quantitative, empirical inquiry to research, develop, and apply a

data-driven model of student programmer behavior based on the analysis of students' edit-by-edit and submission-by-submission actions as they work on programming activities.

A number of prior projects have already made important progress in this direction. In predicting student success (e.g. midterm or final grade), however, prior approaches are weakened by only providing access to part of the total picture. Further, accurate predictions require data collected over a significant period of time. Combining and refining complementary existing lines of inquiry can address these weaknesses and **develop a model of student behavior that has predictive power at the level of the individual assignment.**

SIGNIFICANCE AND IMPACT

A generally applicable strategy for data-driven assessment of learning is a critical need. This white paper proposes the research, development, and evaluation of an infrastructure that combines existing productive, ongoing lines of research regarding the study of novice programmers as they develop software solutions. This effort will combine best-of-breed data collection capabilities regarding (1) student interactions with their code and programming environment as well as (2) their use of tools that support the automatic assessment of their code testing and code correctness. This idea is unique from other work in the study of novice programmers in that it does not focus only on code-authoring or code-assessment, but instead allows us to mine a dataset that provides a comprehensive picture of novice programmer behavior. From this data, we can then develop and validate a model of student programmer behavior and learning improvement that is both data-driven and predictive.

Developing a data-driven model of student programmer behavior that allows deep investigations of student learning gains will be a significant step forward for computing education research. **Researchers** will have access to an unprecedented data stream that captures programmer behavior at both the program edit- and assignment submission-levels. This will enable new empirical investigations into the learning behaviors and the impact of pedagogical changes. **Educators** will have access to tools that allow them to directly evaluate the success of educational interventions they apply in the classroom. **Students** will benefit from tools that are known to support the learning of programming from the start of their learning cycle to the end. **Departments and institutions** will have a framework that supports the automatic collection of data to support assignment- and course-level metrics commonly found in accreditation procedures around the world.

In addition to providing a direct, empirical evaluation strategy for assessing classroom interventions, addressing this research question will also open up new lines of research in terms of data mining and modeling. More specifically, it will then be possible to investigate a predictive, diagnostic model of student performance that will allow educators to automatically identify individuals who need additional support or who are at risk of failing. Initial explorations undertaken as part of prior research suggest we can identify this instructional need *while students work on their current assignment*, which is a significant difference from prior research. Existing work in this area is limited to post hoc characterizations of performance through data analysis, rather than *in vivo* data-driven diagnosis and prediction.

United States interests in such a research strategy stem from the national need for supplying greater numbers of well-trained computing professionals. By enabling new lines of computing education research and classroom evaluation techniques, computing education practices in the U.S. can be materially improved. Improving the delivery of computing education will enable educational institutions to better meet industry's needs for well-trained computing professionals.

BUILDING ON PREVIOUS WORK

A number of research projects exist that collect data regarding student programming. These can be roughly divided into two categories: projects that focus on monitoring student development efforts, typically by instrumenting the IDE used to edit, compile, and run code, and those that focus on analyzing and assessing student work that has been submitted for a grade. While both strategies have strengths, our project captures the entire picture, and provides a quantitative foundation for evaluating both student behavior as well as the effectiveness of teaching interventions.

Instrumenting development tools to collect data as students edit, compile, and run their work provides a number of advantages. It allows the researcher to see when students begin work, how their work progresses over time, what (compile-time and run-time) errors they encounter, how many times they try to fix an error, and more. The primary limitation is that it is usually too difficult to relate this data stream to a meaningful notion of progress toward completing an assignment, or to connect it to any analysis of the quality or correctness of the (partial) solution under construction.

Nevertheless, evaluating data collected from students as they develop programs has produced useful results. Studies of novice programmers suggest that syntactic issues can slow or even halt forward progress in learning [14][2], that even novices prefer to maintain the syntactic correctness of code as they edit [13], and error message quality can impact how quickly students correct syntactic errors in their code [17][20]. By examining students' edits from one compilation attempt to the next, Jadud et al. have replicated results indicating that compilation behavior can predict student success by quantifying where, when, and what kinds of syntax errors students encounter when learning to program in Java [11][22]. Although these studies were conducted in a variety of languages (Alice, Visual Basic, Java, and Racket), they are all related in their use of automatic protocols for the collection of data regarding novice programmers.

Recently, this kind of automatic collection of behavioral data has been increasingly employed in an attempt not just to understand specific aspects of novice programmer behavior, but also to predict or otherwise recognize high- or low-performing students. Tools like GRUMPS provide a character-by-character view of a programmer's activity [24], while tools like Gauntlet, CodeWrite, CodingBat, and Retina operate at a much higher level (compile-by-compile or submission-by-submission) [7][19][21][2]. While the research findings from these kinds of analyses are critical, they are not employed in "real time" to support instructors in understanding student performance from one assignment to the next. An important aspect of our proposed work is to enable continuous, real-time prediction of which students are at risk of failing to produce a successful solution to the assignment on which they are currently working.

Unlike IDE-based data collection, automated grading software usually provides a way to assess functional correctness and (in some cases) code quality. The primary limitation is that such tools typically have access only to the work students have (voluntarily) submitted for assessment. As a result, they do not provide information about a student's work habits before submission or record any information about the dynamic events a student encounters, such as compile-time and run-time errors. Nevertheless, Edwards et al. have leveraged one grading tool, Web-CAT, to accurately model student success based on when they begin work on an assignment (as denoted by a student's first submission to Web-CAT) and the student's success in developing code that passes one or more unit tests that form the core of Web-CAT's automatic assessment engine [5].

Web-CAT provides students with early, automated feedback on solutions they submit to assignments, encouraging students to submit early and often. Further, Web-CAT allows instructors to grade students based on how well they test their own code, making it feasible to allow students unlimited submissions to assignments. For each submission a student makes, information about the time of the submission, adherence to required coding standards, presence of required documentation, number of student tests written, number of student tests passed, statement coverage achieved, branch coverage achieved, and the size of the submission (number of classes, methods, statements, and branches, both for the solution and for the student-written tests), are all collected for analysis.

While much prior work on automated grading exists [9][12][10][8], prior systems also typically focus on assessing whether or not student code produces the correct output. Web-CAT, on the other hand, is typically used in a way that focuses on assessing the student's performance at testing his or her own code, and on generating concrete, directed feedback to help the student learn and improve. Such a tool allows educators to give assignments that require test suites to be submitted along with code [18]. Ideally, students should be able to "try out" their code-in-progress together with their tests early and often, getting timely feedback each time.

In addition to its wide dissemination as an automated grading system, Web-CAT's impact on student learning has already been studied [3][4]. Students produce higher quality code using Web-CAT, with a 28% reduction in the number of bugs per thousand lines of student-written code (KSLOC), on average.

While using Web-CAT and writing their own tests, the **top 20%** of students in the experimental evaluation achieved defect rates of approximately **4 defects per KSLOC or better**, which is comparable to most commercial-quality software written in the United States. Of the students in the control group who were not required to turn in their own tests and were not evaluated on their own testing behavior, none achieved this level of performance, with the best scores reaching only 30 defects/KSLOC. Students who wrote their own tests were also more likely to turn in their assignments on time and achieve higher scores on programming assignments, both of which are significant results at $\alpha = 0.05$. Web-CAT is uniquely qualified as an automated grading system because of its impact on student learning, and because of its fully customizable data analysis and reporting subsystem [1]. This uniquely powerful capability is integral to providing the kind of data access and manipulation capabilities necessary for continuous data-driven assessment.

SUMMARY

The research question posed in this white paper identifies a critical educational research need: **How can computing educators evaluate student learning gains as they learn to program?** No framework for automatically collecting and analyzing student programming behavior—from authoring through assessment—exists today. By providing for complete, beginning-to-end data tracking and analysis of student programming activities, it is possible to significantly impact student learning, educator practice, and the assessment of instructional methods in the computer science classroom. Most importantly, this infrastructure will allow us—within days rather than weeks or months—to identify students who need targeted assistance as they learn to program.

REFERENCES

- [1] Allevato, A., Thornton, M., Edwards, S.H., and Perez-Quinones, M.A. Mining data from an automated grading and testing system by adding rich reporting capabilities. In *Educational Data Mining 2008: 1st Int'l Conf. Educational Data Mining, Proceedings*. Montreal, Quebec, Canada. June 20-21, 2008, pp. 167-176.
- [2] Denny, P., Luxton-Reilly, A., Tempero, E., and Hendrickx, J. Understanding the syntax barrier for novices. In *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education (ITiCSE '11)*. ACM, New York, NY, USA, 2011, pp. 208-212.
- [3] Edwards, S.H. Improving student performance by evaluating how well students test their own programs. *J. Educational Resources in Computing*, 3(3): 1-24, September 2003.
- [4] Edwards S.H. Using software testing to move students from trial-and-error to reflection-in-action. In *Proc. 35th SIGCSE Technical Symposium on Computer Science Education*, ACM, 2004, pp. 26-30.
- [5] Edwards, S.H., Snyder, J., Allevato, A., Perez-Quinones, M.A., Kim, D., and Tretola, B. *Comparing effective and ineffective behaviors of student programmers*. In *Proceedings of the Fifth International Computing Education Research Workshop*, ACM, New York, NY, 2009, pp. 3–14.
- [6] Fenwick, J. B., Norris, C., Barry, F. E., Rountree, J., Spicer, C. J., and Cheek, S. D. Another look at the behaviors of novice programmers. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, ACM, New York, NY, 2009, pp. 296–300.
- [7] Flowers, T., Carver, C., and Jackson, J. Empowering students and building confidence in novice programmers through Gauntlet. *Frontiers in Education*, vol. 1, 2004, pp. T3H/10-T3H/13.
- [8] Isaacson, P.C., and Scott, T.A. Automating the execution of student programs. *SIGCSE Bulletin*, 21(2): 15-22.
- [9] Isong, J. Developing an automated program checker. *J. Computing in Small Colleges*, 16(3): 218-224.

- [10] Jackson, D., and Usher, M. Grading student programs using ASSYST. In *Proc. 28th SIGCSE Technical Symp. Computer Science Education*, ACM, 1997, pp. 335-339.
- [11] Jadud, M.C. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the 2nd International Workshop on Computing Education Research*, ICER '06. ACM, New York, NY, USA, 2006, pp. 73–84.
- [12] Jones, E.L. Grading student programs—a software testing approach. *J. Computing in Small Colleges*, 16(2): 185-192.
- [13] Ko, A. J., Aung, H., and Myers, B. A. Design requirements for more flexible structured editors from a study of programmers' text editing. In *Extended Abstracts CHI 2005: Human Factors in Computing Systems*, Portland OR, April 2-7, 2005, pp. 1557-1560.
- [14] Ko, A. J., Myers, B. A., and Aung, H. Six learning barriers in end-user programming systems. In *IEEE Symposium on Visual Languages and Human-Centric Computing*, Rome, Italy, September 26-29, 2004, pp. 199-206.
- [15] Lister, R. CS EDUCATION RESEARCH: The naughties in CSEd research: a retrospective. *ACM Inroads* 1, no. 1 (March 2010): 22–24.
- [16] Lister, R., Clear, T., Simon, Bouvier, D.J., Carter, P., Eckerdal, A., Jacková, J., Lopez, M., McCartney, R., Robbins, P., Seppälä, O., and Thompson, E. Naturally occurring data as research instrument: Analyzing examination responses to study the novice programmer. *SIGCSE Bull.* 41, no. 4 (January 2010): 156–173.
- [17] Marceau, G., Fisler, K., and Krishnamurthi, S. Mind your language: On novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ONWARD '11. ACM, New York, NY, USA, 2011, pp. 3–18.
- [18] Melnik, G., and Maurer, F. The practice of specifying requirements using executable acceptance tests in computer science courses. In *Companion to the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, ACM, 2005, pp. 365-370.
- [19] Murphy, C., Kaiser, G., Loveland, K., and Hasan, S. Retina: Helping students and instructors based on observed programming activities. *SIGCSE Bull.* 41, no. 1 (March 2009): 178–182.
- [20] Nienaltowski, M.-H., Pedroni, M., and Meyer, B. Compiler error messages: what can help novices? In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education (SIGCSE '08)*. ACM, New York, NY, USA, 2008, pp. 168-172.
- [21] N. Parlante. Nifty reflections. *SIGCSE Bull.* 39(2):25–26, 2007.
- [22] Rodrigo, M.M.T., Tabanao, E., Lahoz, M.B.E., Jadud, M.C. Analyzing online protocols to characterize novice Java programmers. *Philippine Journal of Science*, 138(2), 177-109, 2009.
- [23] Simon, Cutts, Q., Fincher, S., Haden, P., Robins, A., Sutton, K., Baker, B., Box, I., Raadt, M.d., Hamer, J., Hamilton, M., Lister, R., Petre, M., Tolhurst, D., and Tutty, J. The ability to articulate strategy as a predictor of programming skill. In *Proceedings of the 8th Australasian Conference on Computing Education - Volume 52*, ACE '06. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2006, pp. 181–188. <http://dl.acm.org/citation.cfm?id=1151869.1151893>
- [24] Thomas, R.C., Karahasanovic, A., and Kennedy, G.E. An investigation into keystroke latency metrics as an indicator of programming performance. In *Proceedings of the 7th Australasian Conference on Computing Education - Volume 42*, ACE '05. Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2005, pp. 127–134.