Intention-Based Scoring: An Approach to Measuring Success at Solving the Composition Problem

H. Chad Lane
Institute for Creative Technologies
University of Southern California
13274 Fiji Way
Marina del Rey, CA 90292
lane@ict.usc.edu

Kurt VanLehn
Department of Computer Science
Learning Research and Development Center
University of Pittsburgh
Pittsburgh, PA 15260
vanlehn@cs.pitt.edu

ABSTRACT

Traditional methods of evaluating student programs are not always appropriate for assessment of different instructional interventions. They tend to focus on the final product rather than on the *process* that led to it. This paper presents intention-based scoring (IBS), an approach to measuring programming ability that requires inspection of intermediate programs produced over the course of an implementation rather than just the one at the end. The intent is to assess a student's ability to produce algorithmically correct code on the first attempt at achieving each program goal. In other words, the goal is to answer question "How close was the student to being initially correct?" rather than the the ability to ultimately produce a working program. To produce an IBS, it is necessary to inspect a student's online protocol, which is defined as the collection of all programs submitted to a compiler. IBS involves a three-phase process of (1) identification of the subset of all programs in a protocol that represent the initial attempts at achieving programming goals, (2) analysis of the bugs in those programs, and (3) rubric-based scoring of the resulting tagged programs. We conclude with an example application of IBS in the evaluation of a tutoring system for beginning programmers and also show how an IBS can be broken down by the underlying bug categories to reveal more subtle differences.

Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computer and Information Science Education—*Computer Science Education*

General Terms

Measurement

Keywords

intention-based scoring, online protocols, novice programming, intelligent tutoring systems, structured programming

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'05 February 23–27, 2005, St. Louis, Missouri, USA
Copyright 2005 ACM 1-58113-997-7/05/0002 ...$5.00.

1. INTRODUCTION

Traditional methods of evaluating student programs tend to involve scoring of the final program produced by a student for a given project. Although such a score is certainly appropriate for classroom assessment, it reveals very little about the *process* that went into creating the program. The final program is also prone to influence from a variety of outside sources, such as a tutor or helpful friends. For researchers interested in isolating how different experimental manipulations affect programming skill in a finer-grained way, a metric that targets students *during* the act of programming would be more appropriate. In this paper, we propose such a metric called *intention-based scoring* (IBS) and describe an application of it in the evaluation of an intelligent tutoring system for novice programmers.

Assessing process is a particularly challenging problem. To do so for programming, one approach is to use a *charette*, which requires that a student solve a programming problem in a lab environment and under a time limit [1, 8]. Because no assistance is available (it is typically given as a test), there is no chance for outside influence. Secondly, since most students are not able to complete the full task within the time limit, the resulting score of the "final" version of the program is a actually a measure of success of the student at some point in the middle of their implementation.

Another approach is to collect a student's *online protocol*, which is defined as all files submitted to a compiler during an implementation [12, 13]. This provides a chain of "snapshots" representing a path through the space of development of a program. Lying between each pair of these intermediate programs are compile attempts, which can be explained by a variety of underlying cognitive activities that programmers engage in during programming [2, 4]. In this paper, our goal is not to provide a cognitively plausible account for these activities, but rather to provide a method for scoring such protocols. We seek to quantitatively answer the question "How close was the student to being initially correct?"

2. PROGRAMMING KNOWLEDGE

The knowledge that underlies programming is tacit: a completed program is a poor representation of the knowledge and skills needed to produce it. Studies that focus on the content and structure of this knowledge generally identify structured "chunks" that achieve a variety of goals, sometimes called *schemata* [9] or *plans* [10]. In terms of such theories, two key problems have been suggested as a way of

understanding what programmers must do to produce a program [3]:

- **Decomposition problem:** identifying the goals and corresponding plans needed to solve the problem.
- **Composition problem:** implementing and assembling these plans such that the problem is solved correctly.

Although both problems are known to be a challenge for novices, the composition problem is particularly difficult because of subtle interactions and complications that can arise when multiple plans are to be merged [12]. For example, when two goals each imply the need for a loop, the programmer must determine if one loop in the proposed solution can be used to satisfy both plans. Complications such as this are common for novice programmers, and since inspecting only the final version of a program will rarely reveal them, a more targeted evaluation approach is necessary.

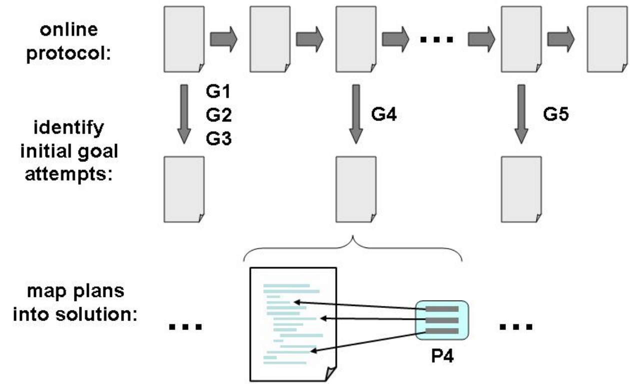

Figure 1: First two stages of producing an IBS.

3. INTENTION-BASED SCORING

IBS derives elements from previous work on identifying bugs in online protocols. Research from this stream has focused on remediation, such as that provided automatically by PROUST [5], as well as on establishing cognitively plausible accounts for how novice bugs are produced [13]. In this section, we present the three phases that are required to produce an IBS.

3.1 Inspecting an online protocol

The first step in producing an IBS is to identify the subset of programs from an online protocol to analyze for bugs. This subset of programs should be the student's *initial attempts* at achieving each goal. A judge begins with the first program submitted and works through the protocol chronologically, checking off goals along the way. This phase is complete when attempts at all goals have been identified or the protocol ends (leaving some goals un-attempted). The process of protocol subset identification is depicted in the top half of figure 1.

The identification process is not always straightforward, however. The first program in a protocol is not always a legitimate goal attempt, for example. Some novices prefer to compile very simple programs to begin, including only things like variable declarations or simple print statements. In cases like this, we ignore such programs and continue searching sequentially for the first substantive attempt at achieving a goal. A related issue is the sometimes fuzzy question of whether or not a program represents an attempt at achieving a goal or not. It is not quite as simple as saying "if any plan component is present, then count it as a goal attempt." For example, some students prefer to declare and initialize all variables at once. This certainly does not imply the student is attempting to implement all plans in which these steps participate.

To handle problems like this, the criteria for selecting programs from a protocol must be agreed upon between multiple judges. In the example we present below, the consensus with the variable declaration issue, for example, was to conclude that by itself, a declaration would not be counted as an attempt at its plan. In other words, more plan components would need to be present than just a declaration or initialization step to count as an outright attempt at that goal. Such issues arise frequently in the subjective tagging of data, which is why it is recommended to tag some subset of the data together under open discussion.

3.2 Bug identification

A critical component of IBS involves the identification and classification of bugs present in a student's protocol. We follow the same two stage process described in [5, 13]. First, the plans being implemented by the student must be identified, and next, compared to the known correct plans of an implementation. Bugs then fall out as differences between these two structures. For IBS, of course, only the plans corresponding to the new goals being implemented at each stage should be considered.

Although there are certainly many ways to characterize the bugs (i.e., plan differences), we adopt here a simplification of the approach taken in [13]. Most generally, an IBS scheme could be constructed from any similar bug classification strategy. Because our goal is not to provide an account of cognitive plausibility, we limit ourselves to categories that relate to solving the composition problem. The top-level categories of bugs in our coding scheme are:

- **omission**: A plan component is missing.
- **malformation**: A component is incorrectly implemented.
- **arrangement error**: A component was placed in the wrong location.

In addition, when inspecting a program, it is also necessary to identify those bugs that are a result of *merging* of plans (e.g., the multiple loop issue mentioned above). We refer to bugs that are *not* a result of confusion between multiple plans as *isolated*. Of course, some bugs can fall under multiple categories. For example, a step can be malformed, out of place, and be a result of confusion between two plans. Because this is a subjective tagging process, it is recommended that multiple judges be used and agreement be checked.

An example of bug identification is shown in figure 2. In this example, the student is attempting to implement a counter plan (shown in the shaded box), but has made three mistakes. First, the incorrect value is used for the initialization step (it should be 0). Second, the increment step is not

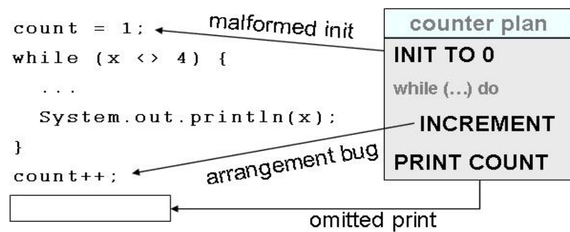

Figure 2: Identifying bugs by plan differences.

placed inside the loop body (an arrangement bug). Finally, there is no print statement (a bug of omission). In this case, the arrangement bug is also considered a plan-merging error since the counter is being integrated into the looping code, which was already in place in this program from a separate plan.

3.3 Scoring

With a bug-tagged protocol, the final step in computing an IBS is to apply a scoring rubric. Although simple bug frequencies could be used, it is less fair since focal steps in plans (i.e., those that are "more" central, such as the increment step of a counter plan) would count the same as less critical components (such as an output statement). To create a rubric, points need to be assigned to the various plan components. Focal steps are weighted more heavily, like updates and conditions, than are supporting steps, like initializations and output statements. This allows us to discount slips (like forgetting to print a value) and highlight errors in critical plan steps. Finally, points for each bug identified during the analysis are taken away from an overall possible score, thereby producing a final intention-based score. In sum, this score represents the accuracy of students' first attempts at achieving programming goals. By looking at points lost from each of the sub-categories (like merge errors or omissions), one can get a better feel for the kinds of errors novices produce.

As an example, for the counter plan in figure 2, one possible assignment could be 3 points for the initialization step, 5 for the increment step, and 2 for the print step. As mentioned, it is best to perform this stage with expert instructors who have experience creating rubrics. In the example, the student might lose 1 point for the incorrect initial value, 3 for the improper location of the increment step, and 2 for forgetting the print step. These partial values need to be agreed upon in the rubric. In sum, this student would receive 4 out of 10 possible points for this attempt at implementing a counter plan.

3.4 Discussion

One difference of IBS with previous work using online protocols is that *all* attempts in a protocol are made available for inspection. Most previous work considered only syntactically correct compile attempts. The reasoning behind "opening" up the protocols in this way comes from the observation that for some students in our protocols, the algorithm intended by the first compile attempt was often different than that in the first syntactically correct attempt. This means that students' algorithms seemed to change, possibly inadvertently, while fixing syntax errors. The very first attempt at assembling an algorithm, in our view, is a more accurate representation of a student's initial impression at how to solve the composition problem. Also, we note that the process is dubbed "intention-based" for two reasons: first, programs are inspected by inferring what goals the student is trying to achieve. Second, when a program statement is not syntactically correct, it is necessary to infer what plan component is being attempted. The line `count + 1;`, for example, is likely an attempt to increment a counter variable. Thus, such a statement is considered equally as correct as a syntactic one.

There are several problems with the IBS method of evaluating programs. First, it is extremely tedious. In no way is IBS intended for regular classroom evaluations – it is only reasonable for use in targeted evaluations that require a fine-grained understanding of student success. Second, the creation of the rubric is subject to the bias of the researcher. In other words, the weighting of the various plan components may indirectly impact the outcome of the study. Finally, in the form presented here, IBS is dependent on a plan-based theory of programming knowledge. The difficulties with this theory, then, are naturally inherited.

4. AN APPLICATION OF IBS

We now turn to an example application of IBS in the evaluation of ProPl ("pro-PELL"), a dialogue-based intelligent tutoring system for beginning programmers [6, 7].

4.1 Experiment

ProPl is intended to help novices do pre-planning of their programs. After the tutoring, students perform their usual, independent implementation. Subjects in the study agreed to allow collection of online protocols. In the sections below, we present the IBS results for the ProPl group when compared to a baseline group of students whoe received no tutoring whatsoever and a control group, who just read the material in a similar environment. The hypothesis being tested was that the program planning skills could be more effectively taught if the interaction occurred as natural language dialogue as opposed to reading alone. In addition to results for two tutored programs (called *Hailstone* and *Rock-Paper-Scissors*), results are also shown for a timed, post-test charette which was not tutored (called *Count/Hold*). The n's were 9, 8, and 9 for the baseline, control, and ProPl groups respectively.

4.2 Training and agreement

After solving the three problems in terms of goals and plans, 15% of all programs were used to train together with two expert judges. After this, another 20% were tagged independently, and then, to confirm agreement, a kappa statistic of .865 was computed on the tags.[1] With consistency of the bug identification procedure confirmed, the remaining protocols were tagged independently.

4.3 Composite intention-based scores

Intention-based scores are shown in table 1 for the three programming projects involved in the study. Some significant and marginally significant differences exist between the

[1]This measure is superior to percent agreement because it factors out agreement by chance. Generally, a kappa value above 0.80 is considered reliable.

problem	baseline	ctrl	ProPl
Hailstone	69.3 (16.4)	79.8 (15.4)	86.1 (9.46)
RPS	67.7 (22.5)	59.5 (18.7)	77.5 (16.4)
CH	n/a	49.1 (26.3)	64.1 (29.8)

Table 1: Composite IBSs, out of 100.

groups. We first consider how the baseline group compared with each of the other groups. For Hailstone, the ProPl students outscored baseline students to a statistically significant level ($t(16) = 2.12$, $p = .0017$) with a very large effect size ($es = 1.03$).[2] The control group also outperformed the baseline group on Hailstone, but not significantly. On the RPS problem, the baseline group outperformed the control group, but the difference is not significant. ProPl students did outperform the baseline students, but again, not to a significant level.

We now turn our attention to the ProPl and control groups. All students in these groups took the same pretest, and so ANCOVAs were used for statistical tests in order to factor out pretest performance. Although ProPl students outperformed the control students on each project, the only significant difference is on RPS. ProPl students were significantly better than control subjects ($F(1, 15) = 7.88$, $p = 0.015$, $es = .96$). On Count/Hold (the untutored posttest charette), ProPl students outperformed those in the control group to a marginally significant level ($F(1, 22) = 3.59$, $p = .072$, $es = .57$).

4.4 Decomposed intention-based scores

The results shown in table 1 are composite scores; that is, the bug categories are lumped together to produce the overall score. To reveal how these points were distributed across the various bug categories, we now proceed to break down the points lost using two sub-categories of bugs.

4.4.1 Merging related errors

In this section, we present the number of points lost related to merging related problems *per opportunity to make an error*. It would be misleading to use the raw points missed. For example, a student who attempts two goals out of a possible five would have far fewer opportunities to produce merging errors than someone who attempts to solve all five. The resulting merge error score would be deceptively low. Similar arguments can be made for plan component omissions and isolated errors. We therefore normalize, and use *total number of goals attempted* throughout the protocol as a denominator. For merge errors, we use the total number of attempted goals, **minus one** because at least two plans are required for a merge error to be possible.

Figure 3 shows the points lost from merging related errors over the three programming problems. For the Hailstone problem, the control group ($M = .38$, $SD = .65$) produced significantly fewer merging related errors than the baseline group ($M = 2.31$, $SD = 1.9$), $t(15) = 2.76$, $p = 0.015$. The ProPl group ($M = .36$, $SD = .45$) performed similarly well when compared to the baseline group ($t(16) = 3.02$, $p = .008$). On RPS, the ProPl group ($M = .19$, $SD = .21$) outperformed both the baseline group ($M = .85$, $SD = 1.0$) to a marginally significant level ($t(17) = 1.98$, $p = .075$, $es = .66$) and the control group ($M = .91$, $SD = 1.1$), $F(1, 15) = 3.71$, $p = .076$, $es = .65$. Finally, on the Count/Hold project, the ProPl group ($M = .07$, $SD = .24$) again surpassed the control group ($M = .61$, $SD = .74$) but this time to a highly significant level ($F(1, 15) = 5.77$, $p = .026$) and with an extremely large effect size ($es = 2.3$).

[2]Effect size was computed using Glass' delta, that is, $\frac{M_{exp} - M_{ctrl}}{SD_{ctrl}}$

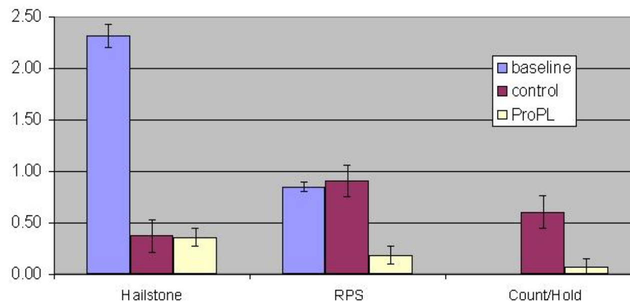

Figure 3: Points lost per plan-merging opportunity. Standard error bars are shown.

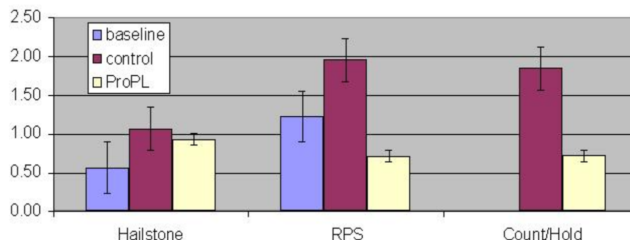

Figure 4: Plan part omission points lost per plan implementation attempt.

4.4.2 Component omission errors

Moving now to omission errors (figure 4), several differences were found to be significant. Interestingly, the baseline group lost *fewer* points in Hailstone for missing plan parts ($M = .57$, $SD = .59$) than the control group ($M = 1.1$, $SD = .53$) to a marginally significant level ($t(16) = -1.85$, $p = .085$, $es = 1.0$). This suggests the baseline group had a greater opportunity for merging errors because they had more plan components to deal with. When compared to the ProPl group, the difference is not significant. This happened again on RPS with the baseline group ($M = 1.22$, $SD = .62$) outperforming the control group ($M = 1.95$, $SD = .67$), but to a significant level ($t(16) = -2.20$, $p = .046$, $es = 1.2$). The control group, in general, seemed to be more forgetful than the other two groups. The ProPl group ($M = .71$, $SD = .63$) also was significantly better than the baseline group on RPS ($F(1, 15) = 15.6$, $p = .0017$, $es = 1.9$). A similar difference appeared on Count/Hold with the ProPl group ($M = .72$, $SD = .62$) losing significantly fewer points than the control group ($M = 1.84$, $SD = 1.13$), $F(1, 15) = 9.22$, $p = .0065$, $es = .99$.

5. DISCUSSION

Several of the detected differences are suggestive of the impact ProPl had on students in the study. First, the higher composite IBSs of ProPl students suggests that they benefited in general from the tutoring by having initial attempts that were closer to correct than students in the other conditions. Looking at the decomposed scores, both groups receiving intervention did equally well over the baseline group on the first program.

The longer term effect, however, seems to favor the ProPl group. In the following two problems (and most importantly, the third, untutored program), ProPl students lost significantly fewer points from errors related to the interactions between plans. That is, dialogue-based tutoring seems to help novices establish a stronger understanding of the issues involved with achieving multiple goals in one program. Given that this is known to be a major difficulty for novices [12], this result bodes well for the efficacy of natural language tutoring to help novice programmers.

The plan part omission results are not as compelling given the relationship between the baseline and control groups. It is surprising that the baseline group *produced more complete plans* than the control group – the control group received some of this information beforehand. Given the very high level of merging-related problems shown in figure 3, there was a price for being more complete. On the positive side, the ProPl group again demonstrates a trend in the correct direction of getting better with respect to plan completeness. Taking these differences to the most general conclusion, this may suggest that they were able to adopt a more abstract view of programming by thinking at the level of plans rather than the line-by-line view normally adopted by novices.

6. SUMMARY AND FUTURE WORK

In this paper we presented intention-based scoring, an approach to assessing online protocols produced by novice programmers. This metric focuses on the *process* of programming by providing a score of a student's ability to solve the composition problem. An IBS is computed by inspecting the first attempt at solving each programming goal over the course of an entire implementation followed by the application of a traditional rubric to those programs. We then demonstrated how to use IBS to evaluate a tutoring system for novices and were able to break the scores down based on the bug categories to evaluate the system in a finer-grained way.

IBS is limited in its application, however. It is bound to a plan-based theory of programming knowledge and is tedious to compute. These problems suggest two lines of future work. The first is to explore other methods of judging the quality of intermediate programs, perhaps in other programming paradigms. The second, and perhaps more interesting, is to use the large amount of tagged data to build automatic classifiers for bug identification.

7. ACKNOWLEDGMENTS

This research was supported by NSF grant 9720359 to CIRCLE, the Center for Interdisciplinary Research on Constructive Learning Environments at the University of Pittsburgh and Carnegie-Mellon University. We also would like to thank Mark Fenner for his help in tagging the protocols, Bob Hausmann's assistance with the experimental setup and statistical analyses, and the anonymous reviewers who provided insightful and useful feedback.

8. REFERENCES

[1] C. Daly and J. Waldron. Assessing the assessment of programming ability. In *Proceedings of the 35th Technical Symposium on Computer Science Education (SIGCSE)*, pages 210–213, Norfolk, VA, 2004. ACM Press.

[2] W. D. Gray and J. R. Anderson. Change-episodes in coding: When and how do programmers change their code? In G. M. Olson, S. Sheppard, and E. Soloway, editors, *Empirical Studies of Programmers: Second Workshop*, pages 185–197. Ablex, Norwood, NJ, 1987.

[3] M. Guzdial, L. Hohmann, M. Konneman, C. Walton, and E. Soloway. Supporting programming and learning-to-program with an integrated cad and scaffolding workbench. *Interactive Learning Environments*, 6(1&2):143–179, 1998.

[4] M. C. Jadud. A first look at novice compilation behavior using bluej. In *16th Annual Workshop of the Psychology of Programming Interest Group (PPIG 2004)*, Institute of Technology, Carlow, Ireland, April 2004.

[5] W. L. Johnson. Understanding and debugging novice programs. *Artificial Intelligence*, 42:51–97, 1990.

[6] H. C. Lane and K. VanLehn. A dialogue-based tutoring system for beginning programming. In *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pages 449–454, Miami Beach, FL, 2004. AAAI Press.

[7] H. C. Lane and K. VanLehn. Teaching program planning skills to novices with natural language tutoring. In S. Fitzgerald and M. Guzdial, editors, *Computer Science Education*. Swets and Zeitlinger, September 2005. Special issue on doctoral research in CS Education.

[8] M. McCracken, V. Almstrum, D. Diaz, M. Guzdial, D. Hagan, Y. B.-D. Kolikant, C. Laxer, L. Thomas, I. Utting, and T. Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year cs students. *ACM SIGCSE Bulliten*, 33(4):125–140, 2001. Report by the ITiCSE 2001 Working Group on Assessment of Programming Skills of First-year CS.

[9] R. S. Rist. Program Structure and Design. *Cognitive Science*, 19:507–562, 1995.

[10] E. Soloway and K. Ehrlich. Empirical studies of programming knowledge. *IEEE Transactions on Software and Engineering*, SE-10(5):595–609, September 1984.

[11] E. Soloway and J. C. Spohrer, editors. *Studying the Novice Programmer*. Ablex Corp., Norwood, New Jersey, 1989.

[12] J. C. Spohrer and E. Soloway. Putting it all together is hard for novice programmers. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, Tucson, Arizona, November 12-15 1985.

[13] J. C. Spohrer, E. Soloway, and E. Pope. A goal/plan analysis of buggy pascal programs. In Soloway and Spohrer [11], pages 355–399.