Teaching the Tacit Knowledge of Programming to Novices with Natural Language Tutoring

H. Chad Lane

Institute for Creative Technologies

University of Southern California, Marina del Rey, CA

Kurt VanLehn

Department of Computer Science, Learning Research & Development Center

University of Pittsburgh, Pittsburgh, PA

Contact Author:

H. Chad Lane

Institute for Creative Technologies

University of Southern California

13274 Fiji Way

Marina del Rey, CA 90292

lane@ict.usc.edu

ABSTRACT

For beginning programmers, inadequate problem solving and planning skills are among the most salient of their weaknesses. In this paper, we test the efficacy of natural language tutoring to teach and scaffold acquisition of these skills. We describe ProPl, a dialogue-based intelligent tutoring system that elicits goal decompositions and program plans from students in natural language. The system uses a variety of tutoring tactics that leverage students' intuitive understandings of the problem, how it might be solved, and the underlying concepts of programming. We report the results of a small-scale evaluation comparing students who used ProPl with a control group who read the same content. Our primary findings are that students who received tutoring from ProPl seem to have developed an improved ability to solve the composition problem and displayed behaviors that suggest they were able to think at greater levels of abstraction than students in the read-only group.

Notes on Contributors:

H. Chad Lane is a Research Scientist at the University of Southern California's Institute for Creative Technologies. He received a B.S. from Truman State University (Mathematics and Computer Science, 1995) and a Ph.D. from the University of Pittsburgh (Computer Science, 2004). His current research involves the application of intelligent tutoring systems and serious games to teach lessons of leadership, inter-personal skills, and cultural awareness.

Kurt VanLehn is a Professor of Computer Science, Professor of Psychology, a Senior Scientist at the Learning Research and Development Center, and Co-Director of the Pittsburgh Science of Learning Center (www.learnlab.org), an NSF Center. He received a B.S. from Stanford (Mathematics, 1974) and a Ph.D. from MIT (Computer Science, 1983). His research interests focus on applications of artificial intelligence to education and cognitive modeling.

1 Introduction

Novice programmers do not generally plan their programs before they attempt to write them. As a result, they often "try to deal with decomposition issues in the middle of coding, instead of planning deliberately in advance." (Perkins, Hancock, Hobbs, Martin, & Simmons, 1989, p.257). This is a Catch-22: a variety of problems arise because novices do not plan, but they lack the knowledge to effectively do so in the first place. In attempts to characterize expert programming knowledge, numerous researchers have suggested the existence of reusable "chunks" of knowledge representing solution patterns that achieve different kinds of goals (e.g., Soloway & Ehrlich, 1984). In this paper, we use the term *schema* to refer to the general form of such chunks, and reserve *plan* to refer to an instantiated version of a schema. In these terms, novice deficiencies in planning are easily explained: they have not yet "built up" a library of schemas from which to draw.

Possessing a library of schemas is only part of the story, however. Two key problems have been suggested as a way of understanding what programmers must solve to produce a program (Guzdial, Hohmann, Konneman, Walton, & Soloway, 1998):

- **Decomposition problem:** identifying program goals and corresponding schemas needed to solve the problem.
- **Composition problem:** implementing and integrating the correct plans such that the problem is solved correctly.

Both are challenging for a novice. Even with a valid decomposition, the composition problem is still very difficult. For example, a major difficulty for novices is the challenge of bringing together separate, but related plans into one program: plans that achieve separate goals often interact in unforeseen ways. Spohrer and Soloway (1985) found that roughly 65% of students' bugs were due to such *plan merging* errors. In this paper, we present ProPl ("pro-PELL", short for PROgram PLanner), a dialogue-based intelligent tutoring system (ITS) intended to address this issue, as well as the decomposition and composition problems in general.

2 Our Solution: Natural Language Tutoring

It is widely acknowledged, both in academic studies and in the marketplace, that the most effective form of education is the professional human tutor. Students working one-on-one with expert human tutors

often score 2.0 standard deviations higher than students working on the same topic in classrooms (Bloom, 1984). In contrast, the best ITS's have effect sizes of only 1.0 standard deviations (Anderson, Corbett, Koedinger, & Pelletier, 1995). In efforts to narrow this gap, a number of dialogue-based tutoring systems have emerged that attempt to mimic the dialogue strategies and communicative patterns of human tutors (Graesser, VanLehn, Rose, Jordan, & Harter, 2001), something most traditional ITS's do not do. In this paper, we propose the use of natural language tutoring to model and support the problem solving and planning activities involved in programming. The aim is to cultivate the development of novices' internal libraries of programming techniques through tutoring. Our central hypothesis is that this knowledge can be learned more effectively if the intervention is done via natural language tutoring as opposed to reading.

2.1 Staged design and the three-step pattern

As mentioned, Spohrer and Soloway (1985) were interested in errors related to the merging of plans. In their analysis, they inspected students *first* syntactically correct programs. Interestingly, they were forced to drop 25 programs (out of about 160) because those students adopted a *staged design* approach, meaning plans were integrated *one at a time* into an evolving solution rather than all at once. Spohrer and Soloway remark that this is evidence of these students dealing with problems of plan merging (p. 731). To explicitly support a staged approach, then, a tutor would need to repeatedly help the student identify goals and how to achieve them, updating the program along the way. This is, in fact, the approach taken in ProPl.

********** INSERT FIGURE 1 ABOUT HERE (3-step) **********

To understand staged design in a dialogue context, we collected a corpus of human tutoring sessions with novices (Lane & VanLehn, 2003). The dialogues seemed to follow a three-step pattern (figure 1) which reveals a progression from a goal, to a general description of how to achieve it (a schema), and ultimately to a plan. The pattern can also be used to guide tutoring in that each part corresponds to a question. For example, to elicit a programming goal (step 1), the tutor normally asks something like "What do you think we should work on next?" Similarly, to elicit a schema, the question might be "How do you think we can do that?" (i.e., achieve some goal).

2.2 Systems for novices

Although an abundant number of systems exist to help novice programmers, most do not *distinctly* address the planning and design phases of programming. Instead, they tend to support implementation-time activities, effectively encouraging "on the fly" planning of programs. Here we discuss a few of the novice programming systems that provide support for separate planning and/or design phases. More thorough reviews of novice programming systems appear elsewhere (e.g., Deek, 1998).

One of the earliest systems to explicitly support a distinct planning phase was Bridge (Bonar & Cunningham, 1988). While using this system, the student designs an English-like solution to a problem via menu selections, then iteratively rewrites this solution into more precise forms with the help of the tutor. Since no conclusive pedagogical benefits regarding the use of Bridge were ever published, it is possible that the use of menus, rather than natural language, did not encourage the sort of deep understanding necessary for improved learning. Another system, MEMO-II (Forcheri & Molfino, 1994), also clearly distinguished planning by providing tools for the construction of abstract specifications used to generate code. Deek (1998) speculates that MEMO-II's abstraction language may have been too complex for novices.

Another relevant system for novices is the GPCeditor (Guzdial et al., 1998), an integrated CAD workbench providing tools for solving the decomposition and composition problems. Students first identify goals and plans needed for a solution, then apply plan composition operators, such as *abut* and *nest*, to assemble the identified plans into a complete solution. An evaluation of GPCeditor revealed that it enabled weaker students to produce quality, working programs, but that higher ability students were unhappy with the added restrictions. Most importantly, positive transfer was observed to a traditional programming environment. Students demonstrated reuse of known plans and of application of the techniques learned to combine them.

The contrast between the approaches taken in GPCeditor and ProPl is striking. Both systems attempt to teach similar skills – that is, to scaffold students while they attempt to solve the decomposition and composition problems. In addition, both systems work from a plan-based theory of programming knowledge. The key difference lies in the approach taken to elicit and convey this knowledge to the student. In GPCeditor, plans are explicitly named and manipulated with the help of the interface, but in ProPl schema-like knowledge is taught *implicitly*, through natural language tutoring.

3 ProPL: A Dialogue-Based ITS for Novices

ProPl ("pro-PELL") is our dialogue-based tutoring system that attempts to model and support simple program planning activities for novice programmers. In this section, we describe the interface and implementation of ProPl and present several example interactions that demonstrate some of the tutoring tactics employed by the system (further detail appears in Lane & VanLehn, 2004).

********** INSERT FIGURE 2 ABOUT HERE (screenshot) **********

3.1 Interface

ProPl runs on any Java-enabled web browser, and is connected to a back end implemented in Lisp that controls tutoring. The interface (shown in figure 2) consists of three windows. The problem statement appears in a mini-browser in the upper left half of the interface, dialogue between the system and student occurs in the chat window in the lower left, and the entire right half of the screen is a dual-tabbed pane: one that holds program *design notes* (shown in figure 2) and another that displays a pseudocode solution. Design notes summarize important observations from the dialogue and act as a *reification* of the tacit knowledge involved in programming. The larger bold-faced entries represent goals, while the lines underneath each goal paraphrase the corresponding schema and plan details. Design notes are pre-authored and linked to the dialogue knowledge sources so the system knows which lines to post and when. The pseudocode is similarly pre-authored and should use familiar words and phrases to make the connections between it, the notes, and the dialogue as clear as possible. An example of a pseudocode solution appears in figure 3.

********** INSERT FIGURE 3 ABOUT HERE (screenshot2) **********

3.2 What happens in a tutoring session

At the beginning of a ProPl session, the design notes and pseudocode panes are empty. Once the student has read the problem statement, the system engages the student in a dialogue intended to confirm understanding of the problem. This initial dialogue is usually very simple, asking the user to simulate the I/O behavior of the desired program. During this interaction, the student is not asked about programming goals or schemas, just the problem task. Once complete, ProPl then moves into repeated questioning as prescribed by the 3-step pattern (section 2.1). As each programming goal is identified,

it is posted in the design notes pane. A "how" question follows and, when answered correctly, aspects of the schemas that achieve these goals are posted as comments beneath the relevant goals. Completing the 3-step pattern, dialogue ensues to describe the needed plan, and finally, the pseudocode screen is updated. Although the student is asked where steps belong, they play no role in actually placing the steps in this version of the system. At the end of the session, the student can review the solution as long as desired.

3.3 Dialogue engine and knowledge sources

ProPl is an application of the Atlas dialogue management system, a domain independent framework for the development of natural language dialogue systems (Rose et al., 2001). Atlas provides two main components: the *Atlas Planning Engine* (APE), a planner for tutorial dialogue management, and the LCFlex parser, a shallow natural language understanding module. To build a tutoring system using Atlas, it is necessary, at a minimum, to provide a plan library to guide tutorial interactions and a semantic grammar and lexicon for LCFlex.

Atlas also prescribes the use of *Knowledge Construction Dialogues* (KCDs). Briefly, a KCD is based on a main line of reasoning that it elicits from the student in a series of questions. If a correct answer is not recognized, a subdialogue is initiated, which can be another KCD or a bottom-out utterance giving away the answer. Different wrong answers can elicit different subdialogues to remedy them, and there is always a generic remedial subdialogue for answers that cannot be recognized as one of the expected wrong answers. Dialogue management in KCDs can be loosely categorized as a finite state model: tutor responses are specified in a hand-authored network and nodes in the network indicate either a question for the student, or a push and pop to other networks. Anticipated student responses are recognized by looking for certain phrases and their semantic equivalents (Rose et al., 2001). ProPl contains 35 top-level KCDs and roughly 100 more remedial KCDs covering three programming problems (about 90 printed pages). KCDs exist for all goals and plans involved in each solution. To create a new problem in ProPl, the domain author must write the KCDs, problem statement, a staged pseudocode solution, and design notes. For one problem in ProPl, this effort required roughly 1.5 hours per KCD (slightly over 50 hours per problem).

3.4 Tutoring tactics

When a student's answer is flawed, ProPl takes action to elicit a correct response. These tactics are authored within KCDs and achieve a variety of tutorial goals. For example, many involve the refinement of vague answers, completion of incomplete answers, and redirection to concepts of greater relevance. There is evidence that encouraging novice programmers to self-explain and critique their impasses improves understanding (Davis, Linn, & Clancy, 1995), and so much of the tutoring of ProPl involves elicitation of ideas.

The two sample dialogues below come from the *Hailstone problem*, an assignment typical of an introductory programming course. It asks the student to write a program to generate a terminating series of numbers according to two simple rules: if the number is even, divide it by 2, otherwise, triple it and add one. This produces sequences of seemingly random lengths. The student is also asked to display the length of the sequence and largest element it contains. Hailstone can be classified as *knowledge-lean*, meaning that it requires minimal specialized domain knowledge. A solution requires the use of several variables, a loop, and conditional statements. It also involves non-trivial merging of several interacting plans and therefore represents a significant challenge for most novices.[1]

********** INSERT FIGURE 4 ABOUT HERE (dialogue1) **********

3.4.1 Simple tactics

Some novice misconceptions are deeply rooted and require multi-turn subdialogues to remediate. In other cases, students only need a slight nudge to get them back on track. Some examples of these shorter, generally single-turn tactics in ProPl include:

- **Content-free pump**: a request for more information, like "Could you say more about that?"
- **Point to the problem statement**: frequently used to elicit a goal
- **Rephrase the question**: used when the expected answer is out of the ordinary or just to give the student another chance.
- **Elicit a simple observation**: usually about the state of the pseudocode to motivate pursuit of a new goal.

[1] Over half of the students in our study spent at least four hours working on their solution.

An example of pointing to the problem statement is shown in figure 4. In this case, the student claims not to know what to do in line 2, but with the suggestion to look at the problem statement in line 3, is able to identify the correct goal. When successful, the simple tactics allow the dialogue to move forward quickly. If a simple tactic fails, the KCDs often shift to an advanced tactic (next section).

3.4.2 Advanced tutoring tactics

Advanced tutoring tactics generally involve multiple turns and require the student to answer several questions along the way. The final step is often some kind of generalization or synthesis of the line of reasoning represented by the KCD (see figure 5, discussed below). Some examples are:

- **Hypothetical situation**: ask the student to imagine a scenario that can cause the program to fail, for example.
- **Elicit an abstraction**: used commonly to elicit a more general answer from an overly specific answer. For example, the tutor might ask "Why should we do that?"
- **Concrete example**: set up a specific situation for discussion.

An appealing property of examples for novices is that they allow the tutor to ask isolated and simple questions that the student can answer. Because knowledge-lean assignments are usually easy to simulate by hand, asking the student to self-reflect during this process can bring out important observations. With the tutor's help, it can even act as a bridge into applicable programming knowledge.

********** INSERT FIGURE 5 ABOUT HERE (dialogue2) **********

There are at least two situations when a reflection tactic is used in ProPl. The first is to help the student think more algorithmically. Figure 5 contains an example of this tactic. When counting a sequence intuitively, most students count *at the end*. The idea of "counting along the way" (i.e., using a counter schema) is often not even considered. In this dialogue, ProPl is using an example to expose the student to this idea, thus better preparing him/her to implement a counter plan in the pseudocode. The second way concrete examples are used by ProPl is to help the student *decompile* his/her existing knowledge and observe the steps. This strategy can be used to elicit goals and schemas. A good example of this comes from the popular game *Rock-Paper-Scissors*: ProPl might ask "What is happening when a player slams his/her fist down?" to elicit the RPS goal of *making a game choice*. Although most students

have played the game before, very few have considered an RPS *algorithm*. Tutoring tactics in ProPL attempt to scaffold precisely this kind of conceptual shift novices must make.

4 Experiment

In this section we describe an experiment to evaluate ProPL completed in the spring of 2004. It was designed to highlight ProPL's use of natural language tutoring.

4.1 Design

Participants were randomly assigned to one of two conditions: ProPL or "click-through" reading. The interface of the control condition's system was identical to ProPL's (figures 2 and 3) but with the dialogue window replaced with canned text. The content was authored such that it mirrored that in the dialogues as much as possible. Both groups viewed identical staged pseudocode solutions and the same design notes. The experiment was designed so that the only real difference between the two groups was the style of interaction.

4.2 Participants

Participants were students currently enrolled in one of three sections of CS0-level introductory (structured) programming at the University of Pittsburgh. Two sections used Java (Computer Science department) and the third used C (Information Science department). All three covered content typical for such courses. Students were admitted on a voluntary basis and only if they had minimal programming experience (no more than one semester). Participants were paid $7/hour for their time and out of roughly 90 students enrolled in the three courses, 33 volunteered. Of these, three were turned away for having too much experience, leaving an initial assignment of n=15 students for each condition. After attrition, the numbers fell to n=12 and n=13 in the ProPL and control groups respectively.

4.3 Materials

The experiment began roughly one month into the semester. Students had learned the basic concepts of computer systems (memory, processors, etc.) and some rudimentary programming concepts including types, variables, operators, and simple I/O. The study spanned nearly six weeks, during which students learned control flow (conditionals and loops) and how to write simple subprograms.

4.3.1 Programming projects

Instructors in the three courses agreed to give the same two assignments needed for this study.

1. **Hailstone**: generate a simple sequence according to rules (section 3.4).
2. **Rock-Paper-Scissors** (RPS): play a "best-of" match, keep track of wins, and pick randomly for the computer

Students were instructed to not start the assignment before coming in for their tutoring sessions. We were able to collect (with permission) copies of all files submitted to the compiler for Java students only. Such data are called *online protocols* and permit a much deeper look into programming ability (Spohrer & Soloway, 1985). The numbers for which we collected online protocols were n=8 (control) and n=9 (ProPl), although final solutions were collected from all participants.

4.3.2 Programming tests

Two written programming tests were developed. The first was given as a pretest to gauge students' general incoming programming competence. The forty-five minute test included conceptual questions, I/O questions, and simple problem solving questions. The posttest was quite different in that it targeted students' planning and algorithm writing skills. More specifically, students were asked to assemble an algorithm given a jumbled collection of steps (including red herrings), organize steps by the goals they help achieve, describe goals achieved by various code segments, identify goals, and finally, solve open-ended decomposition questions. Seventy-five minutes were allotted for students to finish the posttest. Students also completed a *charette* (a timed lab assignment) at the end of the study. The problem, called *Count/Hold*, asked the student implement an intuitively simple, but strategic dice game between the user and a computer player. Students were given two hours to work in our lab with only a textbook and class notes if desired. Online protocols were collected for all participants.

4.3.3 Survey

At the end of the study, students were given a survey consisting of 15 questions. These targeted students' attitudes about the software and how it impacted them during and after their implementations. Several questions also asked about their use or recollection of the design notes versus the pseudocode. A five-point Likert-type scale (1=strongly disagree to 5=strongly agree) was used to score each statement on the survey.

4.4 Procedure

Students began by signing a consent form and answering a background questionnaire. They then took the pretest and used the system to prepare for their first class project (Hailstone). Next, they left our lab and completed that assignment as they normally would over the next week and half, then did the same to prepare for the next assignment (RPS). Notetaking was forbidden during tutoring sessions to prevent programming from degenerating into a pseudocode translation task. For extra practice, students were tutored on one more project, but not asked to implement it. Finally, they returned for the posttests and final questionnaire.

4.5 Intention-based scoring

As mentioned, the purpose of collecting online protocols was to gather a magnified view of how well students were able to solve the decomposition and composition problems. Evaluating this kind of data is difficult and researchers are now beginning to develop tools for this purpose (Jadud, to appear). Simple measures, like raw compile counts and time spent between compiles, are rough and do not necessarily correlate with programming ability. We required a method to quantitatively assess an entire protocol, and so intention-based scoring (IBS) was developed (Lane & VanLehn, 2005). IBS combines elements of goal/plan analysis (Spohrer, Soloway, & Pope, 1989) with traditional rubrics used for scoring programs (such as those in McCracken et al., 2001). Given a protocol, there are three main steps:

1. identify the *first* attempt at each goal
2. compare the implemented plans with correct plans for each
3. using a rubric, assign points to the differences (bugs)

The process is "intention-based" because the judge decides which compile attempts constitute a goal attempt, which steps are correct (ignoring syntax), and which bugs are present. As in goal/plan theory, bugs are defined as differences between the student's code and the solutions represented by the plans. A score is produced by giving point values to every possible bug category, and an overall score is generated from a combination of these. In short, an IBS represents the success of a student's first attempt at achieving each goal.

The top-level categories of bugs in the coding scheme are *omission* (forgetting a step), *arrangement* (placing a step in the wrong location), and *malformation* (an internally flawed step). On top of these,

a step can also be incorrect because of a *merge* error; these are the result of complications arising from integrating separate plans into the same program (see section 2.1). By looking at points lost from each of the bug categories, rather than as a whole, one can generate a more precise picture of novice difficulties (this is done in section 5.2.3).

5 Results

The primary objective of our research was to test the efficacy of natural language tutoring to teach the knowledge that enables simple program planning. Our hypothesis was that students would learn decomposition and composition skills more effectively via natural language tutoring as opposed to reading the same material.

5.1 Pretest

On the pretest covering programming competence, scores were similar for the ProPL group ($M = 68.9$, $SD = 19.2$) and control group ($M = 67.5$, $SD = 18.7$), $t(24) = .155$, $p = .88$, allowing us to safely conclude our assignment to conditions was fair with respect to incoming ability.

5.2 Programming projects

The programming data analyzed in this study covered three programming projects: Hailstone, RPS (section 4.3.1), and Count/Hold (the untutored posttest, see section 4.3.2). Below, we first report final scores on these projects followed by the intention-based results.

********** INSERT TABLE 1 ABOUT HERE (scores) **********

5.2.1 Final program scores

The final scores for the three programming projects involved in this study are shown in left half of table 1. Programs were graded independently by two experienced instructors using traditional rubrics (e.g., McCracken et al., 2001) and with very high agreement ($r(83) = .852$, $p < .0001$). Although the ProPL group scored higher than the control group on two of the projects, none of differences are statistically significant. This is not surprising given that final versions of the programs reveal little about how students arrived at those solutions.

5.2.2 Composite intention-based scores

Intention-based scores are designed to reveal more about the student's process by measuring the correctness of their first attempts at each programming goal. Because we did not have access to online protocols for all students, the number of subjects was reduced to $n = 9$ for the Hailstone and RPS projects in each condition. Inter-rater agreement was high in the bug identification phase of IBS ($\kappa = .865$, see Lane & VanLehn, 2005). The right half of table 1 shows the IBS results.

Although ProPl students outperformed control students on each project, the only significant difference occurred in RPS. ProPl students ($M = 77.5$, $SD = 16.4$) were significantly better than control subjects ($M = 59.5$, $SD = 18.7$), $F(1, 15) = 7.88$, $p = 0.015$, $es = .96$.[2] On Count/Hold, ProPl students ($M = 64.1$, $SD = 29.8$) also outperformed those in the control group ($M = 49.1$, $SD = 26.3$), but to a marginally significant level ($F(1, 22) = 3.59$, $p = .072$, $es = .57$).

5.2.3 Decomposed intention-based scores

The IBS results shown in table 1 are composite scores; that is, they combine the various bug categories into one score. An IBS can tell more of the story by breaking it down according to the bug categories described earlier. To understand how well students were able to work with plans, we now report points *lost* related to plan-merging and plan-part omissions (see section 4.5). It would be misleading to use the raw points missed, however. For example, a student who attempts two goals out of a possible five would have far fewer opportunities to commit merging errors than someone who attempts to solve all five. The resulting loss due to merge errors would be deceptively low. Our solution was to look at points lost *per opportunity to commit that error* by using *total number of goals attempted* as a denominator.[3]

********** INSERT FIGURE 6 ABOUT HERE (merging) **********

Figure 6 shows the points lost from merging related errors over the three programming problems. On RPS, the ProPl group ($M = .19$, $SD = .21$) outperformed the control group ($M = .91$, $SD = 1.1$) to a marginally significant level ($F(1, 15) = 3.71$, $p = .076$, $es = .65$). On Count/Hold (the posttest), the ProPl group ($M = .07$, $SD = .24$) again surpassed the control group ($M = .61$, $SD = .74$) but this time to a significant level ($F(1, 15) = 5.77$, $p = .026$) and with an extremely large effect size ($es = 2.3$).

********** INSERT FIGURE 7 ABOUT HERE (omission) **********

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

[3] For merge errors, we use the total number of attempted goals -1 because at least two plans are required for a merge error to be possible.

Next, looking at students' ability to produce complete plans, figure 7 shows points lost from plan part omission errors. Students in the ProPl group ($M = .71$, $SD = .63$) lost significantly fewer points than the control group on RPS ($M = 1.95$, $SD = .67$), $F(1, 15) = 15.6$, $p = .0017$, $es = 1.9$. A similar difference appeared on Count/Hold with ProPl students ($M = .72$, $SD = .62$) losing significantly fewer points than the control group ($M = 1.84$, $SD = 1.13$), $F(1, 15) = 9.22$, $p = .0065$, $es = .99$.

5.3 Written Posttest

The written posttests were graded by three experienced programming instructors. Each of the six multi-part questions was scored on a scale of 0 to 5 (0 = no attempt, 5 = excellent) giving a maximum possible score of 30. The first five tests were graded together to refine the grading rubrics. Agreement on the remaining exams between the graders, as calculated by a linear regression, was extremely high for each pair ($p_{1,2} = .95$, $p_{2,3} = .94$, $p_{1,3} = .91$, all $r(19) < .001$). To calculate student scores, we took the mean score of the three graders.

Overall, students in the ProPl group ($M = 19.3$, $SD = 4.38$) scored higher than those in the control group ($M = 17.4$, $SD = 6.14$), but this difference is not statistically significant ($F(1, 23) = 2.04$, $p = .17$). By omitting the code comprehension questions (the only ones *not requiring* program planning skills), we found that ProPl students ($M = 17.1$, $SD = 3.75$) again outperformed those in the control group ($M = 15.1$, $SD = 5.1$) to a marginally significant level ($F(1, 23) = 3.00$, $p = .097$, $es = .392$). Some differences were found on individual questions. Regarding algorithm assembly from jumbled steps, ProPl students ($M = 3.39$, $SD = 1.2$) scored higher than control students ($M = 2.56$,$SD = 1.4$), which is significant ($F(1, 23) = 5.72$, $p = .026$, $es = .61$). On code organization by plans, ProPl students ($M = 3.44$, $SD = .70$) similarly were better than control students ($M = 2.95$, $SD = .84$), $F(1, 23) = 24.6$, $p < .001$, $es = .58$. ProPl students ($M = 3.83$, $SD = 1.1$) scored nearly a half a point higher than the control students on goal identification ($M = 3.36$, $SD = 1.2$), but the difference was not found to be significant ($F(1, 23) = 1.0$, $p = .33$). No significant differences existed on the remaining three questions.

********** INSERT TABLE 2 ABOUT HERE (survey) **********

5.4 Survey results

A subset of the survey results are shown in table 2. From these results, it seems that ProPL students believed they understood the material less, focused on the design notes to a greater degree than pseudocode, found debugging easier "because of the use of the system," and that they had more influence on the pseudocode as it was being constructed during tutoring than students in the control group. In reality, neither group had any influence on the pseudocode as it was authored ahead of time. Students in both groups indicated a desire to use the system again in the future, for their own benefit and with no payment, but the difference is not significant between groups (first line, table 2). There is a possible ceiling effect with this question due to the natural bias of using volunteers as participants.

Within the groups, there are also significant differences in questions that reveal how the content of the tutoring was perceived by the students. Two items on the survey asked to what degree they tried to remember the design notes and the pseudocode after their sessions (recall that notetaking was forbidden). Within the control group, students claimed they tried to remember pseudocode ($M = 4.8$, $SD = .62$) more often than design notes ($M = 4.0$, $SD = 1.0$), $t(22) = -2.14$, $p = .044$. Within the ProPL group, the inverse occurred. That is, pseudocode ($M = 4.5$, $SD = .52$) received a lower ranking than design notes ($M = 4.8$, $SD = .39$), but only to a marginally significant level ($t(22) = 1.77$, $p = .090$).

5.5 Discussion

In this section we attempt to synthesize and interpret the results from the previous section. Most importantly, students who were tutored by ProPL seemed to demonstrate *enhanced skill at solving the composition problem.* ProPL students:

- made fewer merging-related errors than students in the other two groups
- omitted fewer plan parts than students in the control group
- scored higher on a written code arrangement task

These differences suggest that dialogue-based interaction led to deeper learning and skill at solving the composition problem.

Turning now to students' ability to work with plans, our data suggests that ProPL *students worked more at the level of schemas and plans* rather than by the line-by-line perspective typically taken by novices. This is supported by three results from the previous section. First, fewer merging related errors

by ProPL students suggests that they developed a heightened sense of the relationships and distinctions between plans. Second, fewer plan part omissions means ProPL students were able to produce more complete plans on their first attempts at implementing them. Third, ProPL students received higher scores on the written posttest problem that asked them to organize steps according to the goals they achieved and the plans to which they belonged. When considered together, these differences suggest that dialogue-based tutoring accelerates the development of the tacit knowledge of programming.

Looking at how the students viewed the help of the two systems on the survey, several differences were found (section 5.4). For one, *students who used the control system believed they understood the material better than students who used* ProPL. This phenomenon is known as "the illusion of knowing" (Glenberg, Wilkinson, & Epstein, 1982) and it seems that dialogue-based tutoring mitigates this effect to a certain degree. It is possible that participation in dialogue is responsible for this difference. Since students receive negative feedback, ProPL may help them develop more accurate self-assessments. In the control condition, students did not have this opportunity unless they took it for themselves.

Another result from the surveys was that ProPL *students preferred the more abstract, conversational style representation of programming knowledge over the more concrete, pseudocode representation.* They rated the value of the design notes higher than the pseudocode when asked what they remembered when thinking back to the tutoring sessions. The inverse rankings were given by control students (but only with marginal significance), pointing to the pseudocode as the preferred representation. Since design notes consist only of short phrases and sentences, it is a more abstract and less structured representation of the solution. In fact, the intent behind using design notes in the first place was to aid in the reification of the tacit knowledge that underlies programming (i.e., goals and schemas). It seems that the use of dialogue raised the comfort levels of those students to use the more abstract representation, at least in their own estimation. This result is in concert with the previously mentioned result that ProPL students seemed to work more on the plan-level rather than line-by-line.

Performance on the open-ended design questions on the written posttest was surprising. Our initial hypothesis was that ProPL's tutoring would lead to improved ability to solve the decomposition problem at an abstract level, but this did not happen. One possible explanation is that students were not directly responsible for the writing or organization of the design notes, in either condition. Therefore, no students had any direct practice producing decompositions written in natural language. This suggests students should be more involved with creating the design notes and if possible, responsible for the content.

Although the experiment was designed to minimize the differences between the ProPL and control groups so that the only difference was the dialogue-based tutoring, there are other potential explanations for these results. First, it is possible that simply encouraging students to articulate their planning ideas ahead of time could produce similar results, although we believe this to be unlikely. Hausmann and Chi (2002) found that a computer-interface providing content-free prompts to students did not succeed in eliciting quality self-explanations, suggesting that more meaningful interaction is needed. Another potential explanation for the differences is time-on-task: students in the ProPL group spent an average of 38.4 minutes in each session while those in the control group spent only 24.8 minutes. This confound is less clear when one considers that learning also likely occurred during the implementation phases, for which students in both conditions spent considerable overall time.

6 Conclusion

We have presented ProPL, a tutoring system for novice programmers designed to help them plan and learn the tacit knowledge of programming. The system is intended to be used by students prior to their pursuit of an independent implementation phase for programming assignments. The goals are to verify the students understanding of the task, elicit goals (the "what") and schemas (the "how") in students' words, and present a staged design of a pseudocode solution. A variety of tutoring tactics are used to do this that all aim to elicit goals and schemas from students. ProPL is intended to support effective pre-planning activities, highlight the important problem solving aspects of the problem, and teach the tacit knowledge of programming by exploiting the properties of natural language tutoring.

Our evaluation aimed to demonstrate the efficacy of natural language dialogue to teach decomposition and composition skills by pitting ProPL against a system that used "click-through" reading to present the same content. We analyzed student behavior both on the tutored projects and posttests covering programming skills and knowledge. Students who used ProPL were frequently better at solving the composition problem in terms of algorithm correctness and presence of bugs. They demonstrated fewer plan merging related errors and plan part omissions in their implementations. In addition, students who used ProPL exhibited behaviors suggestive of thinking at the level of schemas and plans rather than line-by-line. To our surprise, no differences were detected on ability to solve the decomposition problem on a written test.

We originally set out to provide a tool for novice programmers to help them prepare for their assign-

ments more effectively. We found that students who used dialogue to do this showed improvements in their algorithm writing skills and their self-reported appeal of more abstract representations of programming knowledge, but found no improvements in student ability to solve the decomposition problem. We believe these results demonstrate some promise of natural language processing technology to aid in the teaching of the tacit knowledge of programming and continue to seek different ways to integrate natural language tutoring into the process of teaching and learning programming.

References

Anderson, J. R., Corbett, A. T., Koedinger, K. R., & Pelletier, R. (1995). Cognitive tutors: Lessons learned. *The Journal of the Learning Sciences*, *4*(2), 167–207.

Bloom, B. S. (1984). The 2 sigma problem: The search for methods of group instruction as effective as one-to-one tutoring. *Educational Researcher*, *13*, 4–16.

Bonar, J. G., & Cunningham, R. (1988). Bridge: Tutoring the programming process. In J. Psotka, L. D. Massey, & S. A. Mutter (Eds.), *Intelligent tutoring systems: Lessons learned* (pp. 409–434). 1988.

Davis, E. A., Linn, M. C., & Clancy, M. (1995). Learning to use parentheses and quotes in Lisp. *Computer Science Education*, *6*, 15–31.

Deek, F. P. (1998). A survey and critical analysis of tools for learning programming. *Computer Science Education*, *8*(2), 130–178.

Forcheri, P., & Molfino, M. T. (1994). Software tools for the learning of programming: A proposal. *Computers & Education*, *23*, 269–276.

Glenberg, A. M., Wilkinson, A. C., & Epstein, W. (1982). The illusion of knowing: Failure in the self-assessment of comprehension. *Memory & Cognition*, *10*, 597-602.

Graesser, A. C., VanLehn, K., Rose, C. P., Jordan, P. W., & Harter, D. (2001). Intelligent tutoring systems with conversational dialogue. *AI Magazine*, *22*, 39–51.

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

Hausmann, R. G. M., & Chi, M. T. H. (2002, Spring). Can a computer interface support self-explaining? *Cognitive Technology*, *7*(1), 4–14.

Jadud, M. C. (to appear). A first look at novice compilation behavior in BlueJ. *Computer Science Education.*

Lane, H. C., & VanLehn, K. (2003). Coached program planning: Dialogue-based support for novice program design. In *Proceedings of the Thirty-Fourth Technical Symposium on Computer Science Education (SIGCSE)* (pp. 148–152). ACM Press.

Lane, H. C., & VanLehn, K. (2004). A dialogue-based tutoring system for beginning programming. In *Proceedings of the Seventeenth International Florida Artificial Intelligence Research Society Conference (FLAIRS)* (pp. 449–454). AAAI Press.

Lane, H. C., & VanLehn, K. (2005). Intention-based scoring: An aproach to measuring success at solving the composition problem. In *Proceedings of the Thirty-Sixth Technical Symposium on Computer Science Education (SIGCSE)* (pp. 373–377). ACM Press.

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y., et al. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4), 125–140. (Report by the ITiCSE 2001 Working Group on Assessment of Programming Skills of First-year CS)

Perkins, D. N., Hancock, C., Hobbs, R., Martin, F., & Simmons, R. (1989). Conditions of learning in novice programmers. In E. Soloway & J. C. Spohrer (Eds.), (pp. 261–279). Norwood, New Jersey: Ablex Corp.

Rose, C. P., Jordan, P., Ringenberg, M., Siler, S., VanLehn, K., & Weinstein, A. (2001). Interactive conceptual tutoring in Atlas-Andes. In *Proceedings of AI in Education 2001 Conference (AIED).*

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

Soloway, E., & Spohrer, J. C. (Eds.). (1989). *Studying the novice programmer.* Norwood, New Jersey: Ablex Corp.

Spohrer, J. C., & Soloway, E. (1985). Putting it all together is hard for novice programmers. In *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics.*

Spohrer, J. C., Soloway, E., & Pope, E. (1989). A goal/plan analysis of buggy Pascal programs. In E. Soloway & J. C. Spohrer (Eds.), (pp. 355–399). Norwood, New Jersey: Ablex Corp.

1. identify a programming **goal**
2. identify a **schema** for attaining this goal
3. **realize** the resulting **plan** in pseudocode
   (a) create pseudocode step(s) achieving the goal
   (b) place the steps within the pseudocode

Figure 1: 3-step pattern.

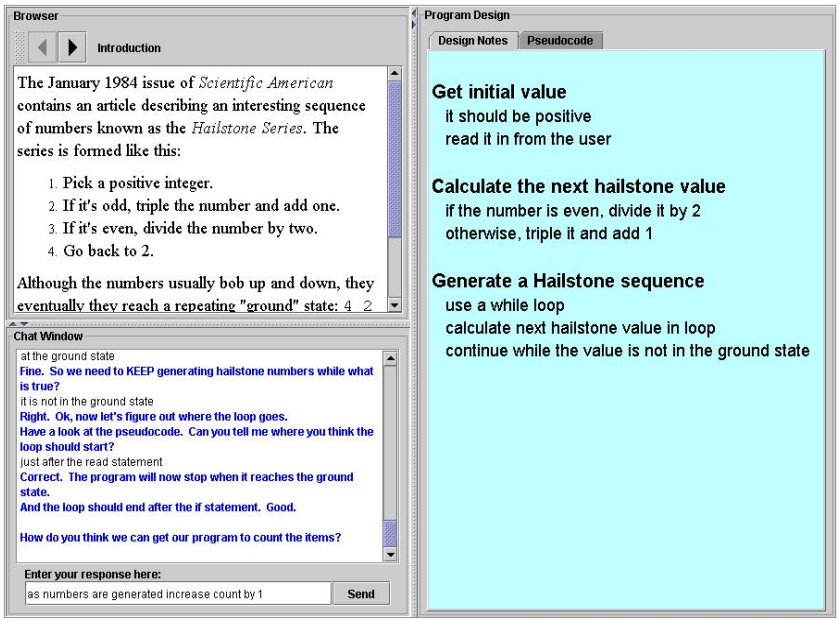

Figure 2: ProPl environment with design notes.

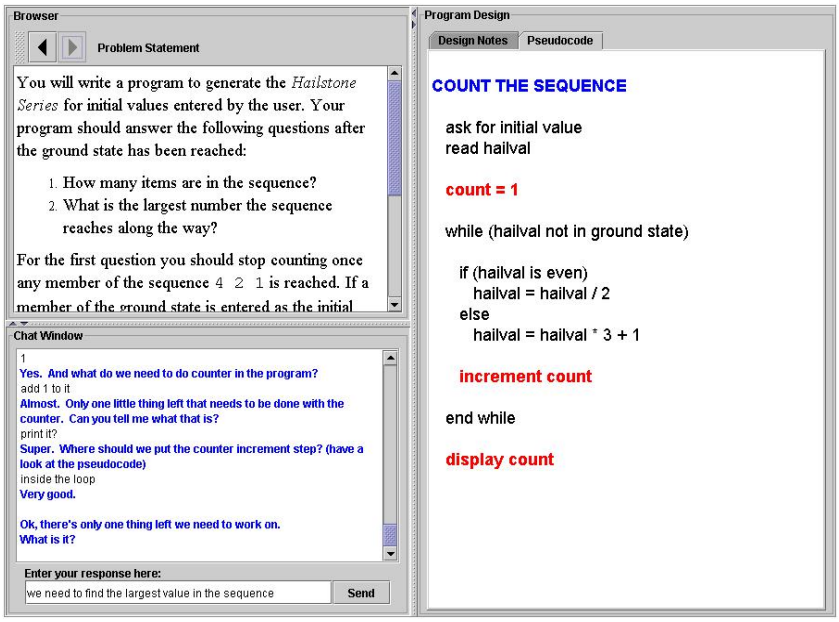

Figure 3: ProPl's Pseudocode screen

1	T	What should we work on now?
2	S	**I don't know**
3	T	You might want to take a look back at the problem statement.
4	T	Give it another shot. What should we work on next?
5	S	**count the items in a sequence**
6	T	Good job.

Figure 4: Eliciting a goal by pointing to the problem statement.

1	T	Let's look at an example to find out how counting might work.
2	T	So, starting with 10. What is the count so far?
3	S	**10**
4	T	Well, we just have 10. The count is therefore just 1 now.
5	T	After 10 in the sequence is 5. What is the count now?
6	S	**2**
7	T	Good job. And after 5 comes 16. What is the count now?
8	S	**3**
9	T	Right. So what is happening to the count?
10	S	**it increments by one each time the loop runs a cycle**
11	T	Yep. How often do we need to do this?
12	S	**until the counter equals 4**
13	T	We will need to add one to the count for each value we see.

Figure 5: Teaching an "on-the-fly" counting strategy.

	Final Scores		IBS	
problem	control	ProPL	control	ProPL
Hailstone	84.3 (19.3)	90.1 (11.4)	79.8 (15.4)	86.1 (9.46)
RPS	76.3 (22.9)	75.8 (23.5)	59.5 (18.7)	77.5 (16.4)
Count/Hold	45.6 (23.5)	48.3 (32.3)	49.1 (26.3)	64.1 (29.8)

Table 1: Final and IBS scores on programming projects. All scores are out of a possible 100 and standard deviations are shown in parentheses.

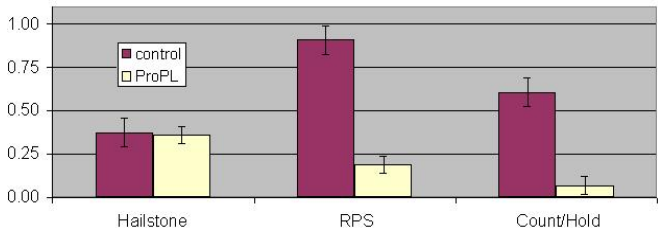

Figure 6: Merging related points lost per opportunity to make such an error on the three programming projects. Standard error bars are shown.

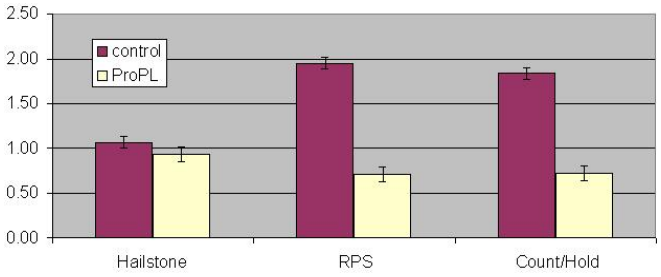

Figure 7: Plan part omission points lost per plan implementation attempt.

survey item	Ctrl (SD)	ProPl (SD)
Would use system again on my own (w/no pay)	4.2 (.94)	4.7 (.65)
I understood the explanations given*	4.7 (.49)	4.1 (.79)
I tried to remember the design notes*	4.0 (1.0)	4.8 (.39)
I tried to remember pseudocode	4.8 (.62)	4.5 (.52)
Debugging was easier because of software*	3.3 (1.1)	4.4 (.79)
I had influence on the pseudocode†	2.8 (1.3)	3.7 (.89)

Table 2: A subset of the survey results comparing mean evaluation scores (1 = strongly disagree, 5 = strongly agree) of students who used ProPl against students who used a click-through, read-only version of the system. * indicates statistical significance ($p < .05$) while † implies marginal significance ($p < .1$), as computed by 2-tailed t-tests.

CAPTIONS:

figure 1: 3-step pattern.

figure 2: ProPl environment with design notes.

figure 3: ProPl's Pseudocode screen.

figure 4: Eliciting a goal by pointing to the problem statement.

figure 5: Teaching an "on-the-fly" counting strategy.

table 1: Final and IBS scores on programming projects. All scores are out of a possible 100 and standard deviations are shown in parentheses.

figure 6: Merging related points lost per opportunity to make such an error on the three programming projects. Standard error bars are shown.

figure 7: Plan part omission points lost per plan implementation attempt.

table 2: A subset of the survey results comparing mean evaluation scores (1 = strongly disagree, 5 = strongly agree) of students who used ProPl against students who used a click-through, read-only version of the system. * indicates statistical significance ($p < .05$) while † implies marginal significance ($p < .1$), as computed by 2-tailed t-tests.