A Dialogue-Based Tutoring System for Beginning Programming

H. Chad Lane & Kurt VanLehn
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260
{hcl,vanlehn}@cs.pitt.edu

Abstract

We present a preventive model of tutoring for novice programming derived from a human corpus and describe our intelligent tutoring system PROPL embodying that model. The system conducts natural language dialogue aimed at eliciting program design ideas from the student prior to their initial solution attempt. Students are asked to identify programming goals and how best to achieve them. Various tutoring tactics are employed to correct flawed responses and refine vague or incomplete answers. PROPL is an application of Atlas, a dialogue management system providing robust sentence-level understanding and a reactive planner to control dialogue. A controlled evaluation is currently underway to assess PROPL's impact on students' programming and problem decomposition skills as well as their general behaviors, beliefs, and attitudes surrounding the tasks of programming.

Introduction

Programming is an exceedingly difficult activity for most beginners. While the reasons for this are multifarious, a significant portion are due to a general lack of problem solving and design skills. Novices generally fail to engage in meaningful planning activities, choosing instead to key in a program as the first step (Pintrich, Berger, & Stemmer 1987). As a result, novices tend to struggle with higher level issues like goal and subgoal structure (Pennington 1987) and the interactions between program logically separate program parts (Spohrer & Soloway 1985).

To address these strategic shortcomings, we have developed PROPL ("PROgram PLanner," pronounced "pro-PELL"), a dialogue-based intelligent tutoring system that engages students in a dialogue *prior* to their attempt to do an implementation. The aim is to model and support the cognitive problem solving and design activities that novice programmers are known to generally underestimate or even bypass altogether. Using only natural language dialogue, students are asked to identify programming goals and then speculate on possible ways to achieve those goals. Thus, contrary the usual path of an unsupervised novice, PROPL promotes a *problem-first* and *goal-oriented* perspective of the tasks of programming.

PROPL is an application of the Atlas dialogue management system (Freedman *et al.* 2000), a domain independent framework for the development of natural language dialogue systems. It provides a reactive planner for dialogue management and a robust sentence-level understanding component, both of which are used extensively by PROPL. We now proceed to lay out the underlying pedagogical foundation for the system, describe its appearance and operation, and describe an ongoing evaluation of its effectiveness.

Tutoring Novice Program Design

A fundamental aspect of most notions of design, especially in the context of novice programming, is *problem decomposition.* Soloway describes this as asking the programmer to "lay the components of the solution on the table, that is, *decompose* the problem and *identify* the solution components" (Soloway, Spohrer, & Littman 1988). These components include:

- **goals** the objectives as indicated by the problem statement
- **schemas** the methods by which the goals are achieved
- **objects** the data items that will be needed in a solution

Ultimately, a programmer must identify these components at some point on their way to a working program. For novices this tends *not* to happen in a distinct planning phase, but rather "on the fly," intermixed with the implementation phase. Inherent problem solving deficiencies therefore compound with shallow programming knowledge resulting in a mystifying and overly complicated experience of questionable pedagogical value.

To assess the efficacy of early tutorial intervention (i.e., "preventive" tutoring), we conducted a human tutoring study of Coached Program Planning (CPP), a style of tutoring designed to elicit problem decompositions from students. Communication occurred over a network, with both tutor and student working in a common environment (nearly identical that of PROPL's, shown in figure 3). On tutored problems and untutored (post-test) problems, CPP was found to have a positive influence on novice programming behaviors in terms of debugging skills and quality of code (Lane & VanLehn 2003).

From this study we also collected a small corpus of 48 tutoring sessions. These dialogues greatly influenced the

design of PROPL by revealing common dialogue patterns, tutor question types, common student answers, tutoring tactics, and conceptual stumbling blocks. In the sections that follow, we describe the key results from this analysis and lay out the resulting cognitive model of tutoring.

Pedagogical context

Our model of tutoring specifically targets *knowledge lean* problems for which the student is being asked to write a traditional structured program.[1] Such tasks require minimal specialized domain knowledge to comprehend and simulate, and are common fodder for introductory programming.

In the rest of the paper, we draw from a particular example of a knowledge-lean task, the **Hailstone problem**. The problem statement appears in the upper left hand corners of both figures 3 and 4. Its solution requires a loop and conditional statements to generate a sequence of numbers. In addition, the student is asked to report the length of the sequence ("counting") and to determine the largest value encountered. For many novices, Hailstone represents a significant challenge because of its nontrivial goal structure and the dependencies between them.

Top-level tutoring pattern

The collaborative goal of the tutor and student in CPP is to build a natural-language-style pseudocode solution to a given problem. This solution plays the role of a design that the student can then use to guide an independent (untutored) implementation. Globally, the tutor's effort is driven by repeated application of a four-step tutoring pattern shown in figure 1. Naturally, the dialogue structure generally coincides with the pattern as well.

1. identify a programming *goal*
2. describe a *schema* for attaining this goal
3. suggest pseudocode *step*s that achieve the goal
4. *place* the steps within the pseudocode

Figure 1: Four-step pattern followed by a human tutor.

Each part of the pattern generally corresponds to a tutor question. For example, to elicit a programming goal (step 1), the tutor normally asks something like "What should we work on now?" To elicit a schema (a "how" question, step 2), the question takes a form similar to "How do you think we can do that?"

From the tutor's perspective, following this line of questioning accomplishes two goals. First, it situates the student at the center of the problem solving task, requiring a suggestion at each point along the way of how best to proceed. Second, it promotes a problem-first, goal-oriented approach to programming that would difficult to instill outside of a tutoring context. We label this kind of tutoring as *preventive* because it permits poor design choices to be discouraged by the tutor and common pitfalls to be detected before they have a chance to propagate into the implementation phase.

[1] Structured (or imperative) languages are those that include the primitives of sequence, conditional execution, and repetition. Pascal and C are two common examples, although most popular languages possess these primitives in some form.

Student linguistic behaviors

A major consideration in building PROPL is dealing with the students' language. As such, we analyzed the student answers in the corpus to better understand the general trends. Based on this analysis, it is clear that students have a number of options when answering design questions. They can identify a problem-specific issue, a relevant general programming notion, or even a low-level programming primitive, and still be considered within the realm of correctness. In understanding these answers, the tutor attempts to identify any "grains of truth" contained in them and build on whatever aspects are productive in the goal of producing a program design.

1. "use a variable"
2. "ok in the loop i'd have a variable starting at 0 that would have a 1 added everytime the loop was executed"
3. "make a function to do it"
4. "keep track of all the elements in the series."
5. "as a number is generated count is increased by one"

Figure 2: Typical student (typed) responses expressing how to implement a counter in a program.

Figure 2 shows several typical responses to the question of how to achieve the counting goal in the Hailstone problem. It is clear that while there is a fair amount of variety, certain properties of the answers are evident. For example, utterances 1 and 3 are overly vague; 2 and 3 describe programming constructs; 2, 4, and 5 include elements of the desired counting schema; and lastly, of these, only 3 was considered wrong by the tutor in the corpus. A more thorough analysis of the language used in the Hailstone dialogues can be found in (Lane 2003).

The corpus reveals that novices are generally able to describe *what* needs to be done (goals) and identify certain aspects of *how* to achieve those goals (i.e., they can describe parts of the required schemas). Not surprisingly, the language they use shows heavy dependence on the problem statement. If the problem statement contains a clearly stated goal or schema, students are better able to articulate it. On the other hand, when the problem statement does not explicitly state a goal or schema, students resort to specific details of the target task or to specific programming concepts. While these conclusions are not particularly surprising, they do have implications for the design of PROPL. Specifically, they speak to the importance of robust understanding and the ability to respond appropriately based on the quality of student answers.

Tutoring tactics

Perhaps the most important trait of any tutoring system is how it remediates student errors and misconceptions. The corpus reveals that students provide a broad range of answers and so, the tutoring module of PROPL needs to handle answers from the very specific to those that are vague or overly abstract.

The overarching theme of the tutor in the corpus is to elicit as much as possible from the student as often as possible. When good answers are provided to top-level questions, then this is easy. However, when the student does not give ideal answers, a subdialogue ensues with the goal of drawing out a better answer. A significant portion is therefore directed at improving student answers. This involves the refinement of vague answers, completion of incomplete answers, and redirection to concepts of greater relevance. Some of the tactics in the corpus include:

- **content-free pump:** "Can you tell me more about that?"
- **refer to problem statement:** "You might want to take a look at the problem statement."
- **elicit requisite observation:** "Does this program generate a sequence?"
- **hint:** "Think about what allows us to repeat something over and over."
- **hypothetical example:** "How much does count go up each time we see a new value?"

When these simple tactics fail (students respond "I don't know" or make no progress beyond their initial answer), the most commonly used tactic in the corpus is to use a concrete example. For example, to elicit the counting schema in the Hailstone problem, the tutor would often walk the student through an example, asking how many values were in the sequence at each point.[2] From this, the desired abstraction (to keep a counter) is much easier to see.

We analyzed the use of concrete examples across the Hailstone dialogues in the corpus. In sum, there were 37 uses of concrete examples, not including obligatory opening examples. Of these, 5 were directly related to the placement of pseudocode and 5 others were used to elaborate the problem statement to the student. This leaves 27 instances, all of which were intended to elicit some answer from the student. To gauge the effectiveness of the strategy, we looked at the within-dialogue success of each of these subdialogues. In other words, each time the tutor engaged an example, the tactic was considered successful if, in the end, the student provided the complete expected answer of the tutor. If the tutor had to give away part or all of the answer, then such instances were labeled as a failure. Of the 27 instances, the student failed to provide the targeted answer only 4 times. Concrete examples therefore played an important role in the corpus, and do the same in PROPL.

[2]Many of students try to perform the count *after* the sequence is generated, which likely matches their intuitive understanding of the task. In this sense, concrete examples permit the tutor to help the student *decompile* this knowledge and view it in more of an algorithmic light.

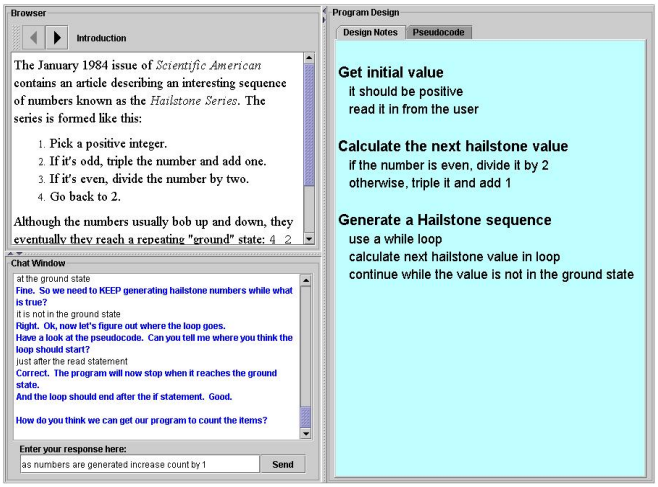

Figure 3: PROPL environment with some design notes.

PROPL: An ITS for Programming

PROPL is dialogue-based tutoring system that attempts to model and support the problem-first, goal-driven style of program planning described in the previous section. In a nutshell, it is intended to introduce the problem to students and help them get started along a productive path.

Interface

The interface runs through any Java-enabled web browser, and is connected to a back end implemented in Lisp that controls tutoring. The interface (shown in figure 3) consists of three primary windows. A mini-browser that holds the problem statement and other supporting materials can be found in the upper left half. Dialogue between the tutor and student occurs in the chat window, found immediately below the mini browser. Finally, filling the entire right half of the screen, is a dual-tabbed pane: one that holds program design "notes" and another that displays a pseudocode solution. For now, the system has complete control over both of these panes.

Design notes are essentially a record of important observations from the dialogue. A sample set of design notes for the first three goals of Hailstone appear in the right half of figure 3. These notes are authored ahead of time by the domain expert and linked to the dialogue knowledge sources so the system knows which lines to post and when. Typically, authors should include all of the programming goals, the need for particular programming constructs, and any other useful observations that can be tied to the dialogues. As each programming goal is correctly identified during the course of a tutoring session, PROPL posts it in the design notes pane. Similarly, as aspects of the schemas that achieve these goals are identified, comments that summarize these observations are posted beneath the relevant goals.

Posting of goals that otherwise generally remain tacit is often referred to as *goal reification*. One reason behind doing this in PROPL is to provide something concrete, but still

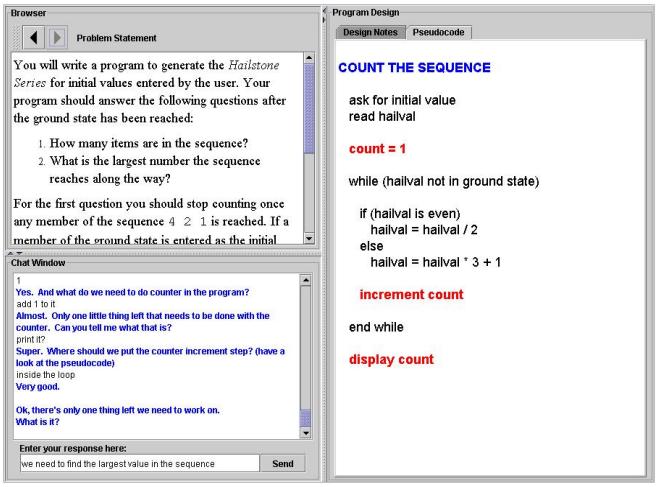

Figure 4: PROPL's Pseudocode screen

T	Ok, great. What should we work on now?
S	**checking the next number**
T	Well, take a look at the pseudocode. How many hailstone numbers does it generate?
S	**just 1**
T	Good. So to do a whole series, what do you think we need to add to the program?
S	**a loop?**
T	That's great. How do you think we can get it to generate a full Hailstone sequence?
S	**put a while loop around the if statement**
T	Right. When will this loop need to stop?
S	**when it reaches the ground state**
T	Correct. So we need to KEEP generating hailstone numbers while what is true?
S	**while it is not in the ground state**
T	Good. Have a look at the pseudocode. Can you tell me where you think the loop should start?
S	**just after the read statement**
T	Yep. The program will now generate a full sequence.

Figure 5: Dialogue between PROPL and a user that identifies the need for a loop in a program.

at an abstract level, representing the planning process as it unfolds. Ultimately, it is hoped that novices who use PROPL will choose to adopt similar habits in future, untutored programming tasks.

The pseudocode solution is presented incrementally as the dialogue evolves. As programming goals and schemas are identified, new pseudocode steps that implement these schemas are introduced into the solution. Figure 4 contains pseudocode just after the student finished describing the counting schema in Hailstone. As with the design notes, the content of the pseudocode is canned. Its inclusion allows the student to view a more precise version of what is contained in the notes pane. In addition, it allows the tutor to refer to the pseudocode (as in figure 5), thus increasing the number of available tutoring tactics.

Sample interaction

Figure 5 presents a sample interaction with PROPL. The dialogue begins just after applying the Hailstone update rules (second goal from top, figure 3). At this point in the solution, the tutor's goal is to elicit the programming goal of generating an entire sequence of numbers. The student is unable to identify this goal, but rather suggests that the "next number" be checked (for evenness or oddness). In other words, the student is thinking about goal of generating more numbers, but has not yet expressed this precise concept. The tutor then asks the student to look at the pseudocode, elicits the applicability of using a loop, and then proceeds to work out the details.

In this example, PROPL must acquiesce to the student's preference for programming primitives by accepting "loop" in place of the more problem-derived "generate a sequence." Indeed, for novices a long discussion about such a difference would likely be viewed as annoying. The ability to recognize these different types of answers and react to them appropriately is due to the robustness of the natural language understanding component of Atlas, to which we now turn.

Dialogue planning and understanding

As mentioned earlier, PROPL is implementation of Atlas (Freedman *et al.* 2000), a domain independent framework for developing natural language dialogue systems. Two main components are provided:

- **Atlas Planning Engine:** (APE) a reactive, "just in time" planner for tutorial dialogue.
- **CARMEL:** a natural language understanding component which includes robust and efficient algorithms for parsing, semantic interpretation, and repair.

To build a tutoring system using Atlas, one must provide access to the user interface, a plan library to guide APE, and semantic mapping rules to guide CARMEL. The plan library should contain operators to handle all desired interactions with the student, including high-level control, producing utterances, performing a GUI action, and engaging in specific tutoring tactics.

Knowledge Construction Dialogues

PROPL's primary knowledge source is a library of Knowledge Construction Dialogues (KCDs) which represent directed lines of tutorial reasoning. They consist of a sequence of tutorial goals, each realized as a question, and sets of expected answers to those questions. The KCD author is responsible for creating both the content of questions and the forms of utterances in the expected answer lists. Each answer is either associated with another KCD that performs remediation or is classified as a correct response. KCDs therefore take on a hierarchical structure and follow a recursive, finite-state based approach to dialogue management.

A graphical representation of a part of PROPL's KCD for eliciting the need for a loop is shown in figure 6. In this

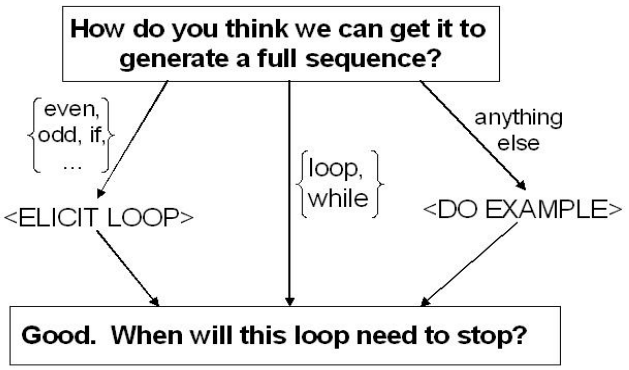

Figure 6: Part of a KCD that elicits the need for a loop

KCD, the context is that the student has recognized the need to generate a sequence, but has yet to establish that a loop is appropriate. The left branch (showing "if", "even", etc.) handles utterances that suggest a return to the extant conditional that handles the even/odd rules. The *Elicit-loop* remediation KCD that is called in this case simply asks the student "Can you think of a way to do that in the program?" This gives the student a second chance to indicate that a loop is needed.[3] The figure also shows that an *anything-else* case is used to handle unrecognized utterances. This is followed when CARMEL is not able to classify an utterance with respect to the expected answer list.

To make KCDs usable by Atlas, they are first compiled into APE operators. This produces a set of readable and modifiable operators, which allow easy integration with any existing plan libraries the system designer may have. To ease the authoring process, Atlas also provides tools for editing KCDs and creating the semantic grammar used by CARMEL (Freedman *et al.* 2000).

The KCD approach is a good fit for PROPL because of the ability to group answers together into classes, and provide specific remediations for each class. CARMEL also provides a *clustering* tool that allows an author to easily add synonyms and alternate answer forms. In addition, authors of KCDs have the option of specifying *answer parts*, meaning that a students answer need to contain mention of each part (or "aspect"). Remediation can be associated within each part, thereby increasing the precision of the ensuing tutoring tactics. In PROPL, answer parts are used to handle top-level responses to schema eliciting questions. Looking back at the answers in figure 2, each of these select different aspects of how counting should work. PROPL looks for the following parts: the use of a variable, initialization, increment step, location of that step, and printing of the result. Naturally, it is highly unlikely all of these will be present in the initial answer, and so the remediation KCDs each try to elicit their respective missing parts.

[3]It was necessary to turn off Atlas' negative feedback, however, because of the abundance of borderline answers such as this. It would not be appropriate to suggest that "go back to the if statement" is a bad response to the question posed by the KCD.

At this time, 11 top-level KCDs have been written for PROPL that conduct dialogue for the Hailstone problem. Over 50 smaller KCDs are used by the top-level KCDs or called by the top-level control operators (next section). In addition, several large remedial KCDs present concrete examples in an effort to elicit the more challenging concepts. Some of these include only brief references to the examples, while others involve complete runs through entire sequences. The decision to enter into these more elaborate remediations is made within the KCDs themselves based on answer classifications and the frequency of unclassifiable input.

Top-level control

While the APE planning operators derived from KCDs govern the lower-level dialogue interactions, the *calling* of KCDs must be controlled by a higher level tutorial component. To perform this control, PROPL includes a host of operators that track the state of the solution and initiate calls to KCDs as needed. The top-level operator includes a list of programming goals that must be satisfied by the dialogue. These goals are considered achieved when the corresponding KCDs for them have completed. In addition, other operators exist that update the notes and pseudocode panes at appropriate times, post corresponding utterances in the chat window, and identify when KCDs have failed to elicit the correct answer from the student.

Handling out-of-scope utterances

To handle unrecognized utterances in KCDs, Atlas provides an "anything else" category. When a student contribution does not map to an expected answer, a generic remedial line of reasoning is followed. KCD authors are encouraged to write "with the intent of sounding natural almost no matter what the student types" (Rose & Jordan 2000). Thus, there is the possibility that good aspects of unexpected answers will be missed and treated as incorrect.

Currently, PROPL includes two simple APE operators that that attempt *KCD recovery*. That is, for utterances not successfully mapped to expected answers, these operators execute dialogue moves that attempt to re-establish the line of reasoning modeled in the KCD. In this case, a *pumping* move (e.g., "Can you say that in a different way?") is generated with the hopes of the student proposing one of the known correct answers in the second chance. There is also an *acknowledgement* operator, a slightly different form of pumping that includes an expression of confirmation when a student utterance involves a concept that is present somewhere in the CARMEL semantic grammar, but not expected at that moment. If this fails, or the utterance is truly out of the scope of the system, the anything-else branch is followed. This extension is particularly appropriate in a design domain due to the "brainstorming" quality of many of the interactions: new ideas can occur at unexpected moments, and often a second chance is all a student needs to provide a good answer.

Related work

A large number of intelligent tutoring systems for programming target specific programming languages (Brusilovsky 1995). While these are suitable for smaller scale tasks and for learning the intricacies the language, many do essentially support on-the-fly planning of programs. There is often no clear distinction made between the traditional phases of programming of understanding, planning, implementing, and debugging. An exception is found with BRIDGE, a system that helps students construct a solution description in natural language (via menus), followed by successive rewrites in more precise forms, ultimately resulting in a working program (Bonar & Cunningham 1988). No conclusive pedagogical benefits regarding the use of BRIDGE were ever published. Another example of such a system is DISCOVER (Ramadhan, Deek, & Shihab 2001), a model-tracing system that supports a restricted form of pseudocode and provides a rich environment for code execution and testing. The only dialogue-based system for programming we are aware of is the Duke Programming Tutor (Keim, Fulkerson, & Biermann 1997) which helped students correct syntax errors in simple Pascal programs via a speech interface. A number of dialogue-based educational systems have emerged recently (e.g., (Rose & Jordan 2000)) for other domains that use more advanced methods of dialogue management. Although it is too early to draw general conclusions about the pedagogical impact of these systems, there is mounting evidence that suggests participation in such dialogues does correlate with learning gains (Core, Moore, & Zinn 2003).

Conclusions and Future work

We have described a model of tutoring and the dialogue-based tutoring system PROPL that is capable of carrying out many of the observed tutoring tactics. The system follows a design-driven pattern that encourages the student to identify goals and describe methods to achieve them. When students are unable to answer correctly, various strategies are used to elicit the desired responses, such as hinting and the use of concrete examples of the targeted task.

A controlled evaluation of PROPL is currently underway. The goal is to determine its effect on novices in terms of both their programming skills and on the perspectives they adopt while programming. By using PROPL, our hope is that novices will (1) learn about how to identify and precisely state programming goals and (2) adopt the general behavior of planning ahead (i.e., thinking about how to achieve the goals they have identified). Various post test measures will be used, including interviews, a closed-lab untutored project, and written problem solving test. A control group will use a non-dialogue-based version of the system that presents the student with the same material provided by PROPL. The aim of the qualitative measures is to understand the reasoning novices used to create their programs and how they perceive their own activities.

For updated information on this evaluation and to try PROPL, please visit:

`http://www.cs.pitt.edu/~hcl/propl`

Acknowledgements

This research was supported by NSF grants 9720359 and 0325054 to Kurt VanLehn at the University of Pittsburgh. We would also like to thank Bob Hausmann and Mike Ringenberg for their comments.

References

Bonar, J. G., and Cunningham, R. 1988. Bridge: Tutoring the programming process. In Psotka, J.; Massey, L. D.; and Mutter, S. A., eds., *Intelligent Tutoring Systems: Lessons Learned*. 1988. 409–434.

Brusilovsky, P. 1995. Intelligent learning environments for programming. In *Proceedings of AI-ED'95, 7th World Conference on Artificial Intelligence in Education*, 1–8.

Core, M. G.; Moore, J. D.; and Zinn, C. 2003. The role of initiative in tutorial discourse. In *10th Conference of the European Chapter of the Association for Computational Linguistics (to appear)*.

Freedman, R.; Rose, C. P.; Ringenberg, M. A.; and VanLehn, K. 2000. Its tools for natural language dialogue: A domain-independent parser and planner. In *Fifth International Conference on Intelligent Tutoring Systems (ITS 2000)*. Springer-Verlag Lecture Notes in Computer Science.

Keim, G.; Fulkerson, M.; and Biermann, A. 1997. Initiative in tutorial dialogue systems.

Lane, H. C., and 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)*, 148–152.

Lane, H. C. 2003. Preventive tutoring in programming: An intelligent tutoring system for novice program design. Dissertation Proposal.

Pennington, N. 1987. Comprehension strategies in programming. In Olson, G. M.; Sheppard, S.; and Soloway, E., eds., *Empirical studies of programmers: second workshop*. Norwood, New Jersey: Ablex Corp. 100–113.

Pintrich, P. R.; Berger, C. F.; and Stemmer, P. M. 1987. Students' programming behavior in a pascal course. *Journal of Research in Science Teaching* 24(5):451–466.

Ramadhan, H. A.; Deek, F.; and Shihab, K. 2001. Incorporating software visualization in the design of intelligent diagnosis systems for user programming. *Artificial Intelligence Review* 16:61–84.

Rose, C. P., and Jordan, P. 2000. Interactive conceptual tutoring in atlas-andes. In *Proceedings of AI in Education 2000 Conference*.

Soloway, E.; Spohrer, J. C.; and Littman, D. 1988. E unum pluribus: Generating alternative designs. In Mayer, R. E., ed., *Teaching and Learning Computer Programming*. 137–152.

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