

A Rule-Based Expert System for Diagnosing Student Errors in a Tutor on Counter-Controlled Loops

Sara Rosenthal
Dept. of Computer Science
Columbia University
New York, NY 10027, USA
sara@cs.columbia.edu

Amruth N. Kumar
Dept. of Computer Science
Ramapo College of New Jersey
Mahwah, NJ, USA
amruth@ramapo.edu

Abstract: We developed a rule-based expert system to diagnose student errors in a tutor designed to help students trace the behavior of counter-controlled loops in C++/Java/C#. The novelty of our approach lies in that we infer bugs in the student's declarative model based on student answer rather than predicting bugs in the student answer based on the student's procedural model. Evaluation of 918 problems showed that with just 27 rules, our abductive approach could correctly diagnose 75% of the cases. Our approach is general, extensible, and scalable.

Keywords: Intelligent tutors, Programming, Expert systems, Diagnosis of student errors, Abductive reasoning.

1 Introduction

Novice programmers find counter-controlled (for) loops hard to understand because of the many components involved in the loop: initialization, condition, update, and loop body. In order to help students learn the behaviour of *for* loops, we developed a problem-solving tutor that can be used for C++, Java or C# [4], available at problets.org. The tutor generates program-tracing problems on 9 different concepts in *for* loops. The tutor generates problems as instances of parameterized templates, and contains 15-20 templates per concept. In the problems, the student is presented a program, and asked to predict the output of the program. The user must enter the output one at a time, and for each output, also select the line of code that generates it. This reified interface reduces the chances that the student can correctly solve a problem without a proper mental model of the domain.

The tutor provides the following feedback: 1) whether the student's answer is correct or not, and 2) narration of the step-by-step execution of the program that explains the correct output/answer [6]. We wanted to extend this feedback by listing the diagnosis of why the student's answer is incorrect, e.g., because the student missed the last iteration of the loop, used an incorrect update value, etc. Such diagnosis can help the student focus on the relevant parts of the step-by-step execution of the program.

But, such diagnosis is hard to perform, especially for *for* loop: the components of the loop can interact in complex ways both due to omission and incorrect commission; and errors can accumulate over multiple iterations of the loop. Therefore, there could be multiple equally plausible diagnoses for an incorrect output and the best diagnosis may at best be a guess.

Given these constraints, we wanted to develop a general, extensible, and scalable solution for diagnosing the output of a *for* loop – general in that we did not have to hand-code all possible diagnoses for each problem in the tutor; extensible in that we could easily extend the diagnostic capabilities to cover additional concepts in *for* loops, and scalable in that we could add additional problems to the repertoire of the tutor without affecting the diagnoses of existing problems. We developed a rule-based expert system because of its extensible and scalable nature. We designed a set of general rules to diagnose the output of *for* loops. We evaluated the accuracy of the diagnosis by analysing the data collected by our *for* loop tutor over three semesters. In Section 2, we discuss how our approach differs from the bug libraries traditionally used in programming tutors. We then describe the syntax and semantics of our rules in Section 3. In Section 4, we describe the algorithm and architecture of our rule-based expert system. We discuss the results from our evaluation of the system in Section 5 and conclude in Section 6.

2 Related Work

With traditional overlay student models, student’s errors can only be characterized as missing elements of the correct knowledge. In order to model student misconceptions, researchers have proposed including bug libraries in the student model (e.g., BUGGY [2]). Later researchers have proposed automating the development of the bug libraries using induction [7], theory refinement [1], etc.

Our work differs from bug libraries in several ways (we limit our comparison to rule-based approaches for diagnosing student errors out of necessity). Most bug libraries have been proposed for procedural knowledge needed to write programs, e.g., PROUST [5], LISP Tutor [9]. The rule-base in our tutor deals with declarative knowledge needed for program tracing (i.e., predicting the output of a program), a classification task similar to that attempted in ASSERT [1].

In almost all prior systems, bug libraries have been incorporated into or used in conjunction with the student model. In our tutor, the rule base, when brought online, will be an independent part of the expert, and not part of the student model. It will only be used to diagnose the student’s answer, and will not be saved as part of the student model. Therefore, we will continue to use an overlay student model, while deriving the benefits of using a bug library. The resulting advantage is that the student model will remain an accurate reflection of the domain knowledge, and can be used for navigation and reinforcement of declarative relationships when opened to scrutiny.

In prior systems, the bug descriptions are of the form:

IF procedural bug e THEN student answer s

i.e., they list how a procedural bug can affect the student's answer. The library of such bug descriptions is used in conjunction with the correct knowledge to *simulate*/predict student behavior. In our tutor, the rules are of the form:

IF student answer s THEN bug e

i.e., for each pattern of a student answer, the rule lists the bug that contributed to it.

The rule-based system is used with the student's answer (not student model) to *infer* the bug in the student's declarative knowledge. Whereas earlier systems use deduction to predict student behavior based on procedural student model, we use abduction to infer declarative student model based on student behavior. Prior systems assume that there are far fewer procedural bugs than there are possible programs resulting from them, and this is true for problems on writing programs. Our tutor assumes that there are fewer patterns of student answer than there are possible errors in declarative knowledge, and this is true for problems on program-tracing. So, both the systems play to the strengths of their domain.

Finally, the rules in our rule base represent generalizations of rather than specific instances of buggy loop behavior. Whereas bugs in bug libraries are applicable to one or some selected problems in a tutor, most of the rules in our rule base are applicable to all the problems – for each problem, the system picks the best among the diagnoses generated by all the rules. This enhances the scalability of our approach.

3 Rules

Each rule contains an IF clause, a THEN clause and an optional WITH clause:

- IF clause lists the conditions that must hold true for the rule to be applied. ELSE IF is a special type of IF clause that denotes a default rule.
- THEN clause contains the template of the explanation that will be produced;
- WITH clause lists the transformations that will be applied to the student's answer and actual answer before they are compared for equality.

The detailed syntax of a rule in BNF-notation is as follows:

```

<rule> → (IF|ELSE IF) <conditionList>
        THEN <explanationTemplate>
        [WITH actualAnswer[<integer>(*|<->integer)] =
         studentAnswer[<integer>(*|<->integer)]]

<conditionList> → <conditionList> & <condition> | <condition>

<condition> → analysis = <analysisType> | result = <resultType>
             | <leftStatement> <relop> <rightStatement>

<analysisType> → Program | Line | Iteration
<resultType> → Correct | Partial | Incorrect
<relop> → < | <= | == | != | > | >=

```

IF clause: The `<condition>` in the IF clause is a conjunctive expression that specifies the type of analysis, type of result and/or comparison of student and actual answers. The **type of analysis** (`<analysisType>` in the BNF rule) refers to the granularity at which the output of the program is analyzed:

- Program: The student's answer is compared with the actual answer for the entire program.
- Line: The student's answer is compared with the actual answer for each line of code that generated the output, e.g., all the student's output for line 12 is compared with all the actual output for line 12. This is made possible because of the reified interface described earlier – the student must identify the line of code that generates each output in the program.
- Iteration: The student's answer is compared with the actual answer for each iteration of the loop, e.g., all the student's output for the first iteration of a loop is compared with all the actual output for the iteration.

The **type of result** (`<resultType>` in the BNF rule) may be - correct, incorrect, or partial. The default result type is partial.

In the **comparison of student and actual answers**, `<leftStatement>` and `<rightStatement>` may contain keywords such as `studentAnswer`, `actualAnswer`, `studentAnswer.SIZE` (the number of entries in the student's answer), `actualAnswer.at1.LINE` (the line number of the first entry in the actual answer), etc.

The `<explanationTemplate>` contained in **THEN clause** is a string that may include variables that can be instantiated, e.g., `$LINE`, which holds the line number of a line of code.

Transformations applied to the student's answer and actual answer, specified in **WITH clause** may be in one of two formats:

- `[m-n]` indicates that the first `m` and the last `n` lines in an answer must be eliminated.
- `[m*n]` indicates that the first `m` lines in the output must be retained, the next `n` lines must be eliminated and this pattern must be repeated.

The transformed student answer is compared with the actual answer and the degree of overlap between them is calculated as the confidence value. The confidence value is used to estimate the correctness of the explanation. It is used to eliminate explanations that may not be helpful to the student because they rely on too much guessing.

Example: Consider the following rule:

IF analysis = Line & template \geq 475 & template $<$ 525
& studentAnswer.SIZE $>$ 0 & actualAnswer.SIZE $>$ 0

THEN You missed the changes to the variable at the end of the loop body on line \$LINE

WITH actualAnswer[] = studentAnswer[1*1]

The IF clause of the rule checks that the output is analyzed by line of code, the problem is generated from a template between 475 and 525, and neither the student answer nor the actual answer is empty (the latter corresponds to the program producing no output). The WITH clause specifies that the student answer will be transformed by removing every other (even) element, as shown below:

Student answer :

```
"9 is printed on line 11"  
"8 is printed on line 11"  
"7 is printed on line 11"  
"6 is printed on line 11"  
"5 is printed on line 11"  
"4 is printed on line 11"
```

Transformed student answer:

```
"9 is printed on line 11"  
"7 is printed on line 11"  
"5 is printed on line 11"
```

If the IF clause evaluates to true, the explanation template in the THEN clause is instantiated (i.e., \$LINE is replaced with the line number), and the explanation "You missed the changes to the variable at the end of the loop body on line 11" is returned along with the confidence value generated by the WITH clause.

Once an explanation is generated, a priority is attached to it based on the analysis type, result type and whether a default (ELSE IF) rule was used. This priority is used to sort and eliminate explanations so that the student receives the best possible explanation. The order of priority from highest to lowest is as follows:

1. Correct Output: Explanations from program-level analysis rules that state that the output is correct are given the highest priority.
2. Iteration Priority: Explanations from iteration-level analysis rules;
3. Line Priority: Explanations from line-level analysis rules;
4. Program Priority: Explanations from program-level analysis rules that diagnose why the output is incorrect;
5. Default Iteration: Explanations from default (ELSE IF) rules for iteration-level analysis;
6. Default Line: Explanations from default rules for line-level analysis;
7. Default Program: Explanations from default rules for program-level analysis are given the lowest priority.

Explanations from default rules are given lower priority because their diagnosis is not specific to the problem/program at hand.

4 The Rule-Based Expert System

The algorithm of the rule-based expert system is as follows:

For each type of analysis (program/line/iteration):

1. Organize the actual and student output for the type of analysis. E.g., for line analysis, use the student/actual output for each line of code; for iteration analysis, use the student/actual output for each iteration of the loop, etc.
2. For each rule applicable to the current type of analysis
 1. Instantiate the rule (e.g., commit values of SIZE, LINE, DATATYPE, TEMPLATE NUMBER, ITERATION, etc.)
 2. Apply the rule to the actual and student answer. If the IF/ELSE IF clause evaluates to true
 - i. Apply the WITH clause, if any. Transform the student and actual answers as specified in the WITH clause.

- ii. Calculate the confidence value by comparing the transformed student and actual answers for equivalence.
 - iii. Calculate the priority based on analysis type, result type and whether a default (ELSE IF) rule was used
 - iv. Instantiate the explanation template in the THEN clause, and return the resulting explanation along with confidence value, and priority.
3. Sort and eliminate explanations
 - Sort the explanations by confidence value
 - Eliminate explanations that have a 0% confidence value

The architecture of the rule-based expert system is shown in Figure 1. The inputs to the system are the student answer, the actual answer and the rules from the rule base. The output of the system is one or more diagnostic explanations for a problem. Most of the operation of the expert system is self-evident. The system uses three tools to aid in evaluating the rules: the instantiator, transformer, and the equivalence tester.

Instantiator replaces variables with context-specific values in the rule templates and explanation templates. Some of the variables it replaces include \$LINE (for line number in the code), ITERATION (for the current iteration of the loop), TEMPLATE (for the template number of the current problem), actualAnswer and studentAnswer. **Transformer** modifies the student answer and actual answer according to the pattern listed in the optional WITH clause.

Equivalence Tester treats as hypotheses, the common syntactic errors that the student is likely to commit while specifying the output of a program. Common syntactic errors include: incorrect case (lower versus upper case) for string outputs, incorrect precision for real outputs, and incorrect white-spaces (space, tab, carriage return) for all outputs. Equivalence Tester finds the best hypothesis according to which the student answer matches the actual answer and appends the hypothesis to the explanation generated by the rule. For example, if a student entered extra carriage returns after each output, the explanation is modified as shown in italics: “Your answer is correct *when spaces are ignored.*” The hypotheses are equivalent to coercion in DEBUGGGY [3].

Sorter/Eliminator: The rule-based expert system returns a set of explanations, each being a 3-tuple of an explanation string, priority and confidence value. Currently, the explanations are sorted in descending order of confidence value, and any explanation with 0% confidence is eliminated. The list of remaining explanations is returned as diagnostic feedback.

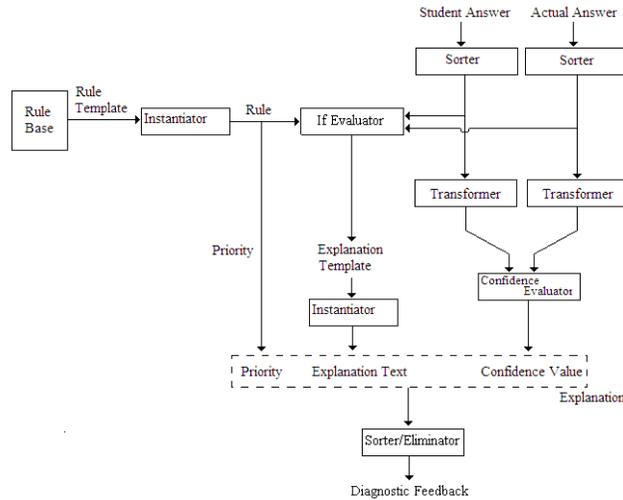


Fig. 1 Architecture of the Rule-Based Expert System

5 Evaluation

We tested the rule-based expert system on the data collected by our tutor on *for* loops over three semesters (69 students). The data consisted of a total of 2119 problems. 1201 problems corresponded to the following four cases:

1. Student Answer = Actual Answer
Explanation: "You identified all the outputs correctly."
2. Student Answer is empty, Actual Answer is empty (corresponding to no output in the program)
Explanation: "Your answer is correct."
3. Student Answer is empty, Actual Answer is not empty
Explanation: "You did not know the answer to the question."
4. Student Answer is not empty, Actual Answer is empty
Explanation: "Your answer is incorrect – you identified non-existent outputs for the code."

Our expert system always diagnosed these cases correctly. Since there was no ambiguity in the diagnosis, we will not consider these problems in the following evaluation.

We analyzed the diagnosis generated by the expert system twice for the remaining 918 problems. During the first analysis, we had 13 rules in the rule base, 6 for program analysis and 7 for line analysis. The outcome was:

- Appropriate diagnosis in 21% of the cases – we determined that these were appropriate by manually verifying the diagnosis against the problem and the student's answer;

- Inappropriate diagnosis in 73% of the cases – either there could have been a more appropriate diagnosis that was *not* generated by the system, or the diagnosis generated by the system was incorrect for the problem and the student’s answer.
- The system did not have adequate information to analyze the remaining 6% of the cases - the student’s answer did not contain enough detail to detect a diagnostic pattern, whether it was analyzed by the system or by a human being. For example, consider the following actual answer and student answer:

Actual Answer:	Student Answer:
“2 is printed on line 9”	“4 is printed on line 9”
“3 is printed on line 9”	
“4 is printed on line 9”	
“5 is printed on line 9”	

There is no reasonable explanation for the student’s output. There is no obvious pattern that can help diagnose the student’s answer.

After the first analysis, we introduced iteration analysis as one of the types of analysis. We modified existing rules and added new rules, resulting in a total of 27 rules - 7 for program analysis, 13 for line analysis, and 7 for iteration analysis. In addition we introduced priority and sorted diagnosis by confidence value. The results of the second analysis were much more encouraging – the outcome was:

- Appropriate diagnosis in 75% of the cases. Of these cases:
 - In 6% of the cases, the most appropriate diagnosis did not have the highest priority. We plan to further refine our priorities to address these cases.
 - In 3% of the cases, the most appropriate diagnosis did not have the highest confidence value because there were other explanations, often generated by default rules, that had a higher confidence. However, this would not preclude the most appropriate diagnosis from being presented to the user, since our system will present diagnoses within a range of confidence values.
- Inappropriate diagnosis in 16% of the cases. We plan to modify a couple of rules and add additional rules to address these cases.
- The system did not have adequate information to analyze the remaining 9% of the cases, since the student’s answer did not contain enough detail to detect a diagnostic pattern.

There is a slight margin of error between the two analyses because of subjective interpretation of the data.

Table 1 lists the appropriateness of diagnoses from the second analysis. The column N lists the number of problems to which the diagnosis applied. Note that the system has little or no room for improvement on diagnoses 3, 4, 5, 7, 8 and 9. It has significant room for improvement on diagnoses 1,2,6,10,11 and 12. Improving priority can significantly improve the accuracy of diagnosis 10. Additional rules may have to be added to the rule-base to improve the accuracy of diagnoses 2, 6 and 12.

TABLE I
RESULTS OF THE SECOND ANALYSIS, CATEGORIZED BY DIAGNOSIS

N	Best	Best	Best	Better	N/A
---	------	------	------	--------	-----

	Overall	Confid.	Priority	Exists	
1. Identified incorrect output – specific error not identifiable					
89	19%	20%	2%	21%	37%
2. Executed the body of the loop as though it had occurred after the loop					
32	56%	0%	0%	44%	0%
3. Iterated the loop too many times					
68	97%	1%	0%	44%	0%
4. Missed iterations of the loop					
94	100%	0%	0%	0%	0%
5. Missed that the loop body is a simple statement					
72	100%	0%	0%	0%	0%
6. Missed that loop variable / parameters are modified within the loop					
52	31%	0%	13%	54%	2%
7. Missed the output statement after the loop					
54	91%	0%	0%	2%	7%
8. Incorrectly executed a nested loop					
118	92%	0%	0%	7%	1%
9. Entered output for the wrong line of code or with incorrect format					
80	91%	4%	3%	3%	0%
10. Switched the order of loop body and update					
79	38%	3%	37%	18%	5%
11. Identified the loop output in incorrect order					
69	23%	0%	17%	12%	48%
12. Initialized the loop variable incorrectly					
36	0%	0%	0%	100%	0%

6 Discussion

Diagnosing the output of a *for* loop is hard, because of the iterative nature of the *for* loop and that its components can interact in complex ways. As a result, there could be multiple equally plausible diagnoses for an incorrect output and the best diagnosis may at best be a guess. Given this complexity and ambiguity, a rule-based expert system is an excellent choice for diagnosing the student output. It allows for the easy addition, modification, and elimination of rules to better analyze a vast range of possible errors that a student can make. We have developed such an expert system and evaluated its accuracy. The system is general, extensible and scalable.

Our expert system is extensible – in order to cover new concepts such as dependent nested loops, we can add new rules to the rule base without affecting the existing rules or their diagnosis. It is scalable. The *for* loop tutor currently contains 203 problem templates. The data that we used to evaluate our diagnostic system used 84 of those templates. Each template can be instantiated into any number of non-identical problems, and the expert system can diagnose any of these problems. Moreover, addi-

tional problems can be added to the tutor on concepts already covered by the rule-base without having to modify the expert system or the rule base.

Finally, our expert system is also general – with only 27 rules, we were able to achieve 75% accuracy in diagnosing the output of 918 problems based on 9 concepts and 84 templates. The novelty of our approach lies in that we infer bugs in the student’s declarative model based on student answer rather than predicting bugs in the student answer based on the student’s procedural model. That we could achieve 75% accuracy with 27 rules is proof of the viability of our abductive rather than deductive approach.

This study was conducted using data collected by a tutor on *for* loops, which is part of a suite of tutors called proplets (proplets.org). Proplets generate problems as instances of templates, grade the correctness of the student’s answer, and provide step-by-step explanation of the correct answer in the form of a fully worked-out example [8]. The rule-based expert system will be incorporated into the tutoring model of *for* loop proplet to supplement the student’s grade with a plausible explanation of why the student’s answer is incorrect. We expect this to improve the effectiveness of proplets at helping students learn *for* loop concepts.

Acknowledgments. Partial support for this work was provided by the National Science Foundation under grant DUE-0817187.

7 References

1. Baffes, P. and Mooney, R., Refinement-Based Student Modeling and Automated Bug Library Construction, *Journal of Artificial Intelligence in Education*, 7, 1 (1996), pp. 75-116.
2. Brown, J. S. and Burton, R. R. (1978). Diagnostic models for procedural bugs in basic mathematical skills. *Cognitive Science*, 2:155-192.
3. Burton, R.B., DEBUGGY: Diagnosis of errors in basic mathematical skills. In D.H. Sleeman and J.S. Brown (Eds.) *Intelligent Tutoring Systems*, Academic Press, New York, 1989: 2, 157-183
4. Dancik, G. and Kumar, A.N., A Tutor for Counter-Controlled Loop Concepts and Its Evaluation, *Proceedings of Frontiers in Education Conference (FIE 2003)*, Boulder, CO, 11/5-8/2003, Session T3C.
5. Johnson, W.L., PROUST: Intention-Based Diagnosis of Errors, Los Altos, CA, Morgan Kaufmann, 1986.
6. Kumar, A.N., Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors, *Technology, Instruction, Cognition and Learning Journal*, 4:1, 2006.
7. Langley, P., and Ohlsson, S. (1984). Automated cognitive modeling. In *Proceedings of the National Conference on Artificial Intelligence*, pages 193-197, Austin, TX.
8. McLaren, B.M. and Isotani, S., When is it best to learn with all worked examples? *Proc. AIED 2011*, Auckland, New Zealand, 2011, 222-229.
9. Reiser, B., Anderson, J. and Farrell, R.: Dynamic student modeling in an intelligent tutor for LISP programming, in *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, A. Joshi (Ed.), Los Altos CA (1985), 8-14.