

Model-Based Generation of Demand Feedback in a Programming Tutor

Amruth N KUMAR
Ramapo College of New Jersey,
505, Ramapo Valley Road, Mahwah, NJ 07430, USA
amruth@ramapo.edu

Abstract. Can a model-based tutor be designed to automatically generate demand feedback for any problem in its domain? Would the resulting feedback be effective enough for the user to learn from? In this paper, we will examine these issues in the context of a tutor for programming. We will propose a two-stage feedback generation mechanism that maintains the principle of modularity characteristic of model-based architectures, and therefore, scalability of the system, while producing coherent demand feedback. Empirical evaluation of our tutors indicates that the generated feedback helps improve learning among users.

Introduction

We have been developing an intelligent tutoring system to help students learn a programming language by analyzing code segments in the language, and predicting their output if they are correct, and debugging them if they are not. Among the six levels of abstraction of educational objectives proposed by Bloom [4], we target application, as opposed to program synthesis, which has been the focus of many earlier works (e.g., LISP Tutor [21], PROUST [12], BRIDGE [6], ELM-ART [8] and Assert [3]). Our work focuses on tutoring programming constructs, and is designed as a supplement to traditional programming projects, as recommended by the whole language approach [19].

We have been using model-based reasoning [10] to model the domain of our tutoring system [13]. One advantage of using model-based reasoning is that the resulting system is more complete and robust in its coverage. This is not necessarily true of rule-based systems (e.g., production rules used in ACT-R theory [1]), which cannot address behaviors unless they have been explicitly encoded into the tutoring system. Similarly, case-based reasoning systems are primarily constrained to the types of cases already entered into the knowledge base [22].

Another advantage of using model-based reasoning is that the tutor is capable of generating answers for problems on its own, i.e., the domain model doubles as the runnable expert module [13]. Could this ability be extended to the generation of feedback, i.e., the feedback is automatically generated by the domain module for any problem, instead of being meticulously hand-crafted for each problem by the designer of the tutor?

Typically, the behaviors of objects in a model are local – each component in a model is responsible only for its behavior; a device that composes other components also composes the behaviors of those components. Can this principle of modularity of model-based architectures be extended to the generation of feedback in a model-based tutor? In other words, can the task of generating the feedback be delegated to the various components in a model, so that each component is responsible for generating feedback

relevant only to its behavior, and any device that composes other components is responsible for composing the feedback generated by those components?

An obvious advantage of using such a modularized architecture for the generation of feedback is that the resulting tutor would be highly scalable - new components could be added to the domain model of the tutor as and when needed, without affecting the feedback generated by any other component. Can coherent feedback be generated by a model-based tutor in spite of this principle of modularity? Would the resulting feedback narrative be effective enough for the user to learn from?

In this paper, we will examine these issues in the context of a tutor for programming. In Section 1, we will describe our model-based representation of the programming domain. In Section 2, we will describe the modularized nature of feedback generation in our model-based tutor. In Section 3, we will list results from the empirical evaluation of our tutoring system to demonstrate that the feedback generated by the system helped students learn. Finally, we will discuss conclusions and future work. Although we currently use C++ in our tutors, the representation and reasoning techniques we propose can be used for any procedural or object-oriented language, including Java and C#.

1. Model-Based Representation of the Programming Domain

We use a component-ontological representation [15,16] in our tutoring system to model the component hierarchy in the domain. As opposed to state-ontological representation (e.g., [7,11,18,23]), in component-ontological representation, the structure and behavior of a device are represented in terms of its components rather than their states, and components, rather than their states are composed to build a hierarchical model of the device. Component-ontological representation promotes better reuse of component models, and simplifies the construction of model hierarchy - it avoids generating the potentially combinatorially explosive number of states that would result from composing the states of the components. Instead, it composes the components themselves.

In the model hierarchy, each device/component is represented in terms of its constituent components. E.g., the program component composes several function components, the function component composes several variable components, and so on. Some interesting characteristics of composition in the programming domain include:

- Structure - It is sufficient to enumerate the composed components. There are no structural interconnections between the components except as specified by the execution of the program.
- Behavior - The composition of the behaviors of the components is dynamically specified, and only specified by the execution of the program. It cannot be specified statically.

We use component-ontological representation throughout the model hierarchy, except at the level of atomic components, where we use state-ontology. We represent the behavior of atomic components as a state transition diagram. Figure 1 shows the state transition diagram for a variable. Its states include declared, assigned and de-allocated. Declaration, Assignment, Input, Initialization, Referencing and Out-of-Scope are actions applied to a variable.

We identify an error E_i for a component as the inability to carry out an action A_i . Each component in the domain model must be capable of generating feedback for every error E_i that might occur in it.

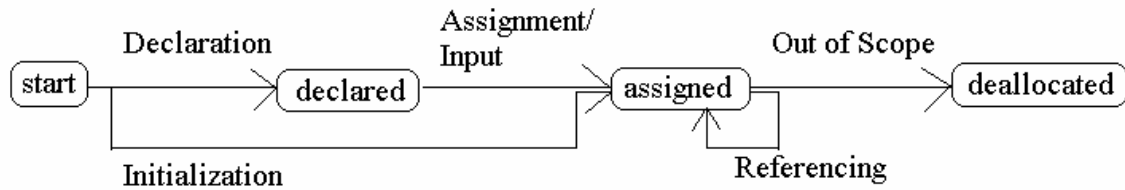


Figure 1. States in the behaviour of a Variable

2. Model-Based Feedback Generation - A Two-Stage Process

Currently, our tutor provides demand feedback [2], i.e., it provides feedback only when the learner demands. Research shows that students who practice with demand feedback are significantly better at far-transfer tasks than those who receive immediate feedback [9,17]. Far transfer is important in programming - students should be able to write programs after practicing with our tutor. According to Guidance Hypothesis [24], demand feedback promotes the use of evaluative tasks, and thereby, long-term retention and transfer. Evaluative tasks include prediction of the behavior and output of a program, necessary to demonstrate an understanding of the program, and error-detection and correction, the mainstays of program debugging. Therefore, demand feedback is very appropriate for a tutor on programming.

We generate demand feedback in our tutor by using reflection (e.g., as in [20]) during the simulation of the model. In other words, as the tutor executes the model of a program, it simultaneously generates explanation about the behavior of the program by explicating its own working. The resulting feedback can not only explain the behavior of the program, but also justify the correct answer.

Our objective is to maintain the principle of modularity common to model-based representations during the generation of feedback, i.e., each component is in charge of generating its own feedback. In order to generate coherent feedback, while maintaining the principle of modularity, we use a two-stage process:

- **Process Explanation:** The C++ interpreter that executes the code generates this explanation in a fashion similar to program tracing. For each line of code, the interpreter identifies the components participating in the line of code, and any state transitions that they undergo as a result of executing the line of code. This includes identifying any side-effect that results from executing the line, i.e., any input, output, or change in the values of variables. Since the lines of code are executed in the order specified by the program, the resulting feedback narrative is coherent.
- **Component Explanation:** The components participating in the line of code being executed generate this explanation. If an action is applied that is not supported by the current state of a component, the component explains why the attempt to apply the action is an error. Table 2 lists the possible errors for a variable on an Action x Event matrix. Each entry corresponds to applying the action denoted by the row title when the variable is in a state denoted by the column title.

If a component is in state S_i when an action A_j applied to it generates an error, the component steps through and generates an explanation for each of the states $S_1 \dots S_i$. It presents the generated lines of explanation in one of two forms:

- **Detailed form:** It explains all the states $S_1 \dots S_i$ in order, and how they culminate in the error when the action A_j is applied.
- **Abbreviated form:** It explains only the last state S_i , and how applying the action A_j results in the error.

Table 1. Errors that occur in a Variable

	declared	assigned	de-allocated
Declaration	<i>Syntax Error</i>	<i>Syntax Error</i>	Code OK
Assignment/Input	Code OK	Code OK	<i>Syntax Error</i>
Referencing	Semantic Error	Code OK	Syntax Error
Out of Scope	Code OK	Code OK	Not Applicable

Component explanation is coordinated with process explanation as follows: during the generation of process explanation, at each line of code, component explanation is obtained from all the components participating in that line of code. Only the explanations from the components that are in an error state are inserted into the process explanation. The abbreviated form of component explanation is used during the generation of demand feedback.

This two-stage process for generating demand feedback can be applied to any domain wherein:

- The behavior of the components can be modeled as a state diagram;
- Aggregation of behavior reflects the aggregation of structure, i.e., those and only those components that constitute a device contribute to the behavior of the device.

2.1 An Example

As an example, consider the following program presented by the tutor as a problem on pointers:

```
void main()
{
    long *referencePointer;
    {
        long size = 5475;
        referencePointer = &size;
    }
    cout << *referencePointer;
}
```

The demand feedback generated for the above code is:

When the program is executed:

The program always starts executing from main().

When the function main() is executed:

Pointer referencePointer is declared on line 4. But, it has not been assigned a value yet

When the nested block starting at line 5 is executed:

Variable size is declared on line 6. It is initialized during declaration to 5475

Pointer referencePointer is assigned to point to size on line 7

The nested block is exited on line 8

Variable size goes out of scope and is deallocated on line 8

An attempt is made to print the value of the variable pointed to by referencePointer on line 9

*But, variable size has already gone out of the scope of its declaration.

*Therefore, referencePointer is a dangling pointer.

The function main() is exited on line 10

Pointer referencePointer goes out of scope and is deallocated on line 10

Whereas **most** of the lines of feedback are generated by process explanation, the lines preceded by asterisk are generated as abbreviated component explanation. Note that it is important for component explanation to include the complete context necessary to understand its feedback independent of any other lines generated by process explanation.

The tutor post-processes the feedback generated by process and component explanations to address the needs of the user:

- **Simulative Feedback:** Since the feedback includes a complete explanation of the behavior of the program, it is used for novices, for the first few problems in a tutoring session, and in instructional (as opposed to problem-solving) mode.
- **Diagnostic Feedback:** The feedback is post-processed to include:
 - The abbreviated explanation generated by components;
 - Process explanation generated corresponding to input and output.

This feedback is used once the student starts making progress towards the educational objectives of the tutor.

Table 2 lists the various options for generating feedback in our model-based tutor. As the table indicates, we plan to use the unabbreviated component explanation to provide prompts for immediate feedback. For each line of the component explanation, questions could be presented to the user to initiate a Socratic dialog that helps the user arrive at the correct answer for the problem.

Table 2. Generating feedback in a model-based tutor

Process Explanation	Component Explanation	Level of Feedback
Included	Included, abbreviated	Simulative feedback
Included selectively	Included, abbreviated	Diagnostic feedback
Not included	Included, not abbreviated	Immediate feedback

3. Evaluation of the Tutor

To date, our tutoring system covers variables, scope, pointers, dynamic allocation, rudiments of function calls, for loops, parameter passing and classes. The system can present problems that involve:

- **Prediction of output:** The learner is expected to predict the behavior and output of a program based on his/her mental model of the program.
- **Identification of bugs:** The tutoring system focuses on those errors that would require a mental model to identify and debug, typically semantic and run-time errors, but also some syntax errors. Most syntax errors trapped by a compiler, such as missing semi-colons and unbalanced braces are not covered by the tutoring system.

In both the cases, solving the problems pre-supposes the availability of a mental model in the learner.

True to our expectation of scalability of component-ontological model-based representation, adding additional topics to our tutoring system has involved adding additional domain components without modifying any existing ones. An example of the *transitive* nature of the scalability of our architecture is that once we had built components for variables and scope, our tutor automatically understood the concept of global variables.

We tested three versions of our model-based tutors - on parameter passing, pointers and *for* loops. Our objective was to evaluate whether users could learn from the feedback that the tutors generated using the two-stage model-based mechanism we proposed. We used controlled tests: the control group received minimal feedback, i.e., whether their answer is correct or wrong, but no explanation; the test group received simulative demand feedback. The protocol involved a written pre-test, followed by practice with the tutor, and a written post-test. The hardness of the pre-test and post-test were the same – they included similar (but not the same) problems, listed in the same order.

Table 3 lists the result of evaluating our parameter passing tutor. We considered student score per attempted question to eliminate practice effect. The effect size was calculated as (post-test score - pretest-score) / standard-deviation on the pre-test. An effect size of one standard deviation for simulative feedback indicates that students could learn from the feedback provided by the tutor. It compares favorably with the result that one-on-one human tutoring improves learning by two standard deviations over traditional classroom instruction [5]. In addition, this improvement in learning is statistically significant ($p < 0.05$).

Table 3. Results of Evaluating the Parameter Passing Tutor, Points per Attempted Question

Feedback		Pre-Test	Post-Test	Effect-Size	<i>p</i>-value
Minimal Feedback (N=15)	Average	2.90	2.97	0.07	0.859
	Standard Deviation	1.09	0.74		
Simulative Feedback (N=14)	Average	2.04	3.31	0.99	0.001
	Standard Deviation	1.28	0.70		

Similarly, when we evaluated our pointers tutor, we found that the scores improved by 44.06% for the test group (N=22) which received simulative feedback, versus 33.84% for the control group (N=16) which received minimal feedback. Both the improvements were statistically significant. On yet another evaluation, we found that the feedback generated by the tutor was effective when compared against printed workbooks [14].

The results from our evaluation of the tutor on *for* loop in Spring 2004 are equally encouraging. We evaluated the tutor against printed workbooks, and evaluated up-counting and down-counting loops separately. The average improvement in the percentage of questions correctly answered from pre-test to post-test and the corresponding 2-tailed *p*-value for the two groups are listed in Table 4. Clearly, students learned from the feedback generated by the tutor, and the improvement was statistically significant.

Table 4. Results of Evaluating the *for* Loop Tutor, Percentage of Questions Correctly Answered

	Workbook Users	Tutor Users
Up-counting Loops	N = 25	N = 24
Average	0.13	0.31
2-tailed <i>p</i> -value	0.08436	0.00003
Down-counting Loops	N = 25	N = 22
Average	0.05	0.33
2-tailed <i>p</i> -value	0.18908	0.00007

4. Conclusions and Future Work

Our work demonstrates that model-based tutors can automatically generate demand feedback by reflection. We proposed a two-stage feedback generation mechanism that maintains the principle of modularity while producing coherent demand feedback. Empirical evaluation of our tutors indicates that the generated feedback helps improve learning among users. Our feedback generation mechanism is scalable.

We plan to extend our work to generate immediate feedback as described in Section 3. Using component-ontological representation, we plan to continue to add components to

the domain model until our tutoring system covers the entire C++ programming language. We plan to extend the tutors to cover Java and C#. Finally, we plan to continue to evaluate our tutors for the quality and quantity of feedback they provide.

5. Acknowledgments

Partial support for this work was provided by the National Science Foundation's Course, Curriculum and Laboratory Improvement Program under grant DUE-0088864.

References

- [1] Anderson, J.R.: Production Systems and the ACT-R Theory. In *Rules of the Mind*. Hillsdale, NJ: Lawrence Erlbaum & Associates, Inc. (1993) 1-10.
- [2] Anderson J.R., Corbett A.T., Koedinger K.R. and Pelletier R.: Cognitive Tutors: Lessons Learned. *The Journal of the Learning Sciences*, Lawrence Erlbaum Associates, Inc. Vol No 4(2) (1995) 167-207.
- [3] P. Baffes and Mooney, R. J.: A Novel Application of Theory Refinement to Student Modeling. *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, Portland, OR, August (1996) 403-408.
- [4] Bloom, B.S. and Krathwohl, D.R.: *Taxonomy of Educational Objectives: The Classification of Educational Goals*, by a committee of college and university examiners. Handbook I: Cognitive Domain, New York, Longmans, Green (1956).
- [5] Bloom, B.S.: The 2 Sigma Problem: The Search for Methods of Group Instruction as Effective as One-to-One Tutoring. *Educational Researcher*, Vol 13 (1984) 3-16.
- [6] Bonar, J. and Cunningham, R.: BRIDGE: Tutoring the programming process, in *Intelligent tutoring systems: Lessons learned*. J. Psozka, L. Massey, S. Mutter (Eds.), Lawrence Erlbaum Associates, Hillsdale, NJ (1988).
- [7] Bouwer, A. and Bredeweg, B.: Aggregation of Qualitative Simulations for Explanation. In: E. Aimeur and K. Koedinger (eds.) *Proceedings of the Workshop on Model-Based Educational Systems and Qualitative Reasoning: The Next Generation*, ITS 2002, San Sebastian, (June 2002) 15-23.
- [8] Brusilovsky, P., Schwarz, E. and Weber, G.: ELM-ART: An intelligent tutoring system on the World Wide Web. *Proceedings of ITS 96 : Third International Conference on Intelligent Tutoring Systems*, Montreal, Quebec, June (1996).
- [9] Corbett, A.T. and Anderson, J.R.: Locus of feedback control in computer-based tutoring: impact on learning rate, achievement and attitudes. *Proceedings of CHI 2001*. ACM 2001. (2001) 245-252.
- [10] Davis, R.: Diagnostic Reasoning Based on Structure and Behavior. *Artificial Intelligence*, 24 (1984) 347-410.
- [11] Koning, K. de., Bredeweg, B., Breuker, J. and Wielinga, B.: Model-based reasoning about learner behavior. *Artificial Intelligence*, Vol 117 (2000) 173-229.
- [12] Johnson, W.L.: *Intention-based diagnosis of novice programming errors*. Morgan Kaufman, Palo Alto CA (1986).
- [13] Kumar, A.N.: Model-Based Reasoning for Domain Modeling in a Web-Based Intelligent Tutoring System to Help Students Learn to Debug C++ Programs. *Proceedings of Intelligent Tutoring Systems (ITS 2002)*, LNCS 2363, Biarritz, France, June 5-8, 2002, 792-801.
- [14] Kumar, A.N.: Learning Programming by Solving Problems, *Informatics Curricula and Teaching Methods*, L. Cassel and R.A. Reis ed., Kluwer Academic Publishers, Norwell, MA, 2003, 29-39.
- [15] Kumar, A.N. and Upadhyaya, S.J.: Component Ontological Representation of Function for Reasoning About Devices. *Artificial Intelligence in Engineering*, Vol 12(4), (1998) 399-415.
- [16] Kumar, A.N. and Upadhyaya, S.J.: Function-Based Candidate Discrimination During Model-Based Diagnosis. *Applied Artificial Intelligence*, An International Journal, Vol. 9(1), (1995) 65-80.
- [17] Lee, A.Y.: Using tutoring systems to study learning. *Behavior Research Methods, Instruments and Computers*, 24(2), (1992) 205-212.
- [18] Mallory, R.S., Porter, B.W. and Kuipers, B.J.: Comprehending complex behavior graphs through abstraction. In Iwasaki, Y. and Farquhar, A. (eds.) *Proceedings of the Tenth International Workshop on Qualitative Reasoning*, AAAI Press, Menlo Park, CA, (1996) 137-146.

- [19] Mann, P., Suiter, P., and McClung, R.: A Guide for Educating Mainstream Students. Allyn and Bacon, 1992.
- [20] Neuper, W.A.: A 'calculemus-approach' to high-school math? In S. Linto and R. Sebastiani (eds.), Proceedings of the 9th Symposium on the Integration of Symbolic Computation and Mechanized Reasoning, Siena, Italy, (June 2001).
- [21] 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).
- [22] Reyes, R.L. and Sison, R.: A Case-Based Reasoning Approach to an Internet Agent-Based Tutoring System. In: Moore, J.D. Redfield, C.L. and Johnson, W.L. (eds.): Artificial Intelligence in Education: AI-ED in the Wired and Wireless Future, IOS Press, Amsterdam (2001) 122-129.
- [23] Rickel, J. and Porter, B.W.: Automated modeling of complex systems to answer predication questions. Artificial Intelligence, Vol 93 (1997) 201-260.
- [24] Schmidt, R.A., Young, D.E., Swinnen, S. and Shapiro, D.C.: Summary knowledge of results for skill acquisition: Support for the guidance hypothesis. Journal of Experimental Psychology: Learning, Memory and Cognition, 15, (1989) 352-359.