Explanation of step-by-step execution as feedback for problems on program analysis, and its generation in model-based problem-solving tutors

Amruth N Kumar
Ramapo College of New Jersey
505 Ramapo Valley Road
Mahwah, NJ 07430

Abstract

We have been developing problem-solving tutors for programming. The tutors target program analysis tasks – evaluating expressions, predicting the output of programs and debugging programs. As part of the feedback, the tutors provide explanation of the step-by-step execution of programs. In this paper, we will present the pedagogy behind providing such explanation. In order to be able to automatically generate such explanation, we will propose using the principles of Model-Based Reasoning to model the programming domain. We will also propose a two-stage algorithm to automatically generate explanation of the step-by-step execution of programs. We will describe two tutors – on parameter passing and for loops - that provide such explanation. We will describe the evaluation of these tutors that confirms that explanation of the step-by-step execution of programs helps students learn to analyze programs.

1. INTRODUCTION

Most of the intelligent tutors for programming have been developed to learn to write programs. Examples include tutors for LISP (e.g., LISP Tutor [Reiser, Anderson & Farrell 85], ELM-ART [Weber & Brusilovsky 01]), Prolog (e.g., VCProlg [Peylo et al 00], Prolog Tutor [Hong 04]), Pascal (e.g., PROUST [Johnson 86], BRIDGE [Bonar & Cunningham 88]), C (e.g., C Tutor [Song et al 97]), C++ (e.g., Assert [Baffes & Mooney 96]), and Java (e.g., JITS [Sykes & Franek 03]).

In contrast, we have been developing intelligent tutors to learn to analyze programs. The problem-solving tasks addressed by our tutors include:

- Evaluating expressions step-by-step;
- Predicting the output of a program line by line;
- Debugging a program – identifying bugs in a program, the program objects to which the bugs apply and the lines of code where they manifest. The primary focus of the tutors is semantic and run-time errors, but some syntax errors are also covered.

Pennington [Pennington 87w] argues that comprehension-demanding tasks such as these play an important role in the development of expertise in programming.

Intelligent tutors for learning to write programs may be classified into two types based on the problem-solving support they provide [du Boulay and Shothcott 87]:

- Tutors that analyze the student’s solution, such as VCProlg [Peylo et al 00], ELM-ART [Weber & Brusilovsky 01], MENO-II [Soloway et al 83], and Prolog Tutor [Hong 04]. ELM-ART also enables the student to test partial solutions with the aid of a language interpreter and provides example-based programming, wherein, the student can request the tutor to show similar problems that had been solved earlier.
- Tutors that analyze the problem-solving process, such as LISP Tutor [Reiser, Anderson & Farrell 85], PROUST [Johnson 86], BRIDGE [Bonar & Cunningham 88], and C Tutor [Song et al 97]. Among these, LISP Tutor is proactive, i.e., it intervenes during the program-writing process, whereas the rest are reactive, i.e., they intervene after the student has written the program.

In this paper, we propose providing explanation of step-by-step execution of a program as problem-solving support in tutors for learning to analyze programs. To this end, we will discuss the pedagogy behind providing explanation of step-by-step execution in section 2. In section 3, we will discuss how
tutors can automatically generate explanation of step-by-step execution of programs. In section 4, we will describe a debugging tutor on C++ pointers, and evaluate it. In section 5, we will describe a tutor on for loop output, and evaluate it.

2. STEP-BY-STEP EXPLANATION FOR LEARNING TO ANALYZE PROGRAMS

The prevalent view of science education is constructivism, which maintains that information cannot simply be transferred from the teacher to the student. Instead the student must construct his or her own meaning [Ben Ari 01]. The most productive way to help a student construct his/her own meaning is through scaffolding, where the teacher guides the student during problem-solving [Wood & Wood 96]. Explanation of the step-by-step execution presented by a tutor provides such scaffolding for learning programming.

An effective mental model of a computer or computation is essential for learning programming. Researchers have attributed students’ poor knowledge of programming to the lack of such a mental model [Perkins, Schwartz & Simmons 88]. Even after a full semester of traditional instruction in Pascal, researchers have observed that students’ mental model, i.e., their knowledge of the conceptual machine underlying programming can be very fuzzy [Sleeman et al 88]. This is compounded by the fact that even when no effort is made to present a view of such a model, students will form their own [du Boulay 89]. Studies in Pascal [Fleury 91], Smalltalk [Holland, Griffiths & Woodman 97] and Prolog [Taylor 90] programming have shown that the mental models built by students on their own are often non-viable. The difference between the “notional” machine mentally constructed by novice programmers and the correct model can contribute to the difficulties encountered by the students [Brusilovsky et al 97, du Boulay, 89].

Step-by-step execution of a program reifies the semantics of execution. Explanation of step-by-step execution helps scaffold the development of the correct mental model and/or repair the non-viable mental model already built by the student. By itself, step-by-step explanation may not be motivating, and therefore, may be pedagogically ineffective. But, since assessment/self-assessment has the potential to enhance meta-cognitive awareness, and serve to highlight the bugs/gaps in the student’s understanding, presenting the explanation after assessment will motivate the student to read the explanation in order to repair his/her understanding of the problem.

Pennington [Pennington 87w, Pennington 87c] investigated the detailed mental representations formed by programmers studying programs written in the imperative style and developed a program comprehension model. The model was derived from earlier models developed and refined for text comprehension [Johnson-Laird 83, Schmalhofer & Glavonov 86, van Dijk & Kintsch 83]. Pennington’s model of program comprehension consists of the following layers, which are progressively more abstract:

- **Surface form** representation of the program text, consisting of the reader’s verbatim memory of the text;
- **Program Model**, which in turn consists of the knowledge of:
  - Operations carried out in the source code, usually in a single statement;
  - Control flow, i.e., the order in which the lines of the source code are executed.
  These are at a low level of abstraction, and are explicitly available in the program text.
- **Domain Model**, which in turn consists of the knowledge of:
  - Data flow, i.e., transformations that are applied to variables as a program executes to change data from input state to output state.
  - Functions, i.e., goals that a program accomplishes. This knowledge is spread over multiple program units. Understanding it may require knowledge of the domain of the program.
  These are at a higher level of abstraction since they are distributed and implicit rather than localized and explicit in a program.

By definition, step-by-step explanation will directly impact the program model. It will help novices make the transition from surface form to the development of the program model.

There are not too many traditional avenues to provide explanation of the step-by-step execution of programs. Instructors typically present programming constructs in class, and do not have sufficient time to explain the step-by-step execution of more than one sample program for each language construct. Textbooks are non-interactive. Due to page limitations, they typically do not include explanation of the
step-by-step execution of programs. In typical program development environments, details of program execution remain hidden, obscuring the semantics of programming languages. Though most program development environments and a few tutoring systems (e.g., [Weber & Brusilovsky 01, Peylo et al 00]) do provide the option of executing programs step-by-step, such execution is not accompanied by explanation that draws attention of the learner to the most salient effects of each step. Without such explanation, step-by-step execution reduces to passive visualization of the values of the program variables, the state of the program input and output, and the value of the instruction pointer. Research shows that visualization must be accompanied by explanation to be pedagogically effective [Naps, Eagan & Norton 00, Stasko, Badre & Lewis 93, Kumar 05]. Therefore, a tutor that textually explains the step-by-step execution of programs would provide unique support for learning, support provided by no other element in the traditional Computer Science learning environment. It would provide such support in a self-paced learning environment. To the best of our knowledge, no tutors have been developed that provide explanation of the step-by-step execution of programs.

2.1 An Example of Explanation of Step-by-Step Execution

In our tutors, we provide explanation of step-by-step execution to help students learn to analyze programs. In this section, we will provide an example of the explanation of step-by-step execution provided by our tutor on debugging C++ pointer programs. Consider the following listing of a C++ program administered by the tutor. It contains a bug that novice programmers find rather difficult to debug – a dangling pointer.

```c++
1. void main()
2. {
3.     int * ptr;
4.     {
5.         int count = 5;
6.         ptr = &count;
7.     }
8.     cout << *ptr;
9. }
```

A tutor could ask the student whether, what, where and why the above code contains a bug:

- **Whether**: The tutor could ask the student whether the above program is correct, or if it contains bugs. This would be a simple yes/no question.
- **What**: The tutor could ask the student to clarify the type of bug, if any, that exists in the code. This reduces the chances of the student randomly guessing the correct answer, since any of a number of different types of bugs could occur in the code.
- **Where**: The tutor could ask the student to identify the line where the bug, if any, exists. This prompts the student to reify the answer. It probes the mental model of the student, since a bug could occur anywhere in the code.
- **Why**: If the identified bug could be caused in more than one way, the tutor could ask the student to clarify the mechanism that resulted in the bug. E.g., a dangling pointer could be caused in one of three ways: when an un-initialized pointer is dereferenced, when a pointer is dereferenced after the dynamic variable to which it points is deallocated, or when a pointer is dereferenced after exiting the scope of the variable to which it points.

In our tutor on C++ pointers, we use a reified interface [Kumar 03a] which asks the student to identify 1) the line in the code where the bug exists, 2) select the bug and finally 3) select the reason for the bug.

It stands to reason that when the student does not solve the problem correctly, the tutor provides an explanation of the step-by-step execution of the program which clarifies all four of the above questions: whether, where, what and why. Such an explanation, as actually generated by our tutor on C++ pointers follows:
When the program is executed:
The program always starts executing from main().
When the function main() is executed:
Pointer ptr 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 count is declared on line 6.
It is initialized during declaration to 5
Pointer ptr is assigned to point to count on line 7
The nested block is exited on line 8
Variable count goes out of scope and is deallocated on line 8
An attempt is made to print the value of the variable pointed to by ptr on line 9
But, variable count has already gone out of the scope of its declaration.
Therefore, ptr is a dangling pointer.
The function main() is exited on line 10
Pointer ptr goes out of scope and is deallocated on line 10

Note that the explanation points out that the bug is on line 9, where a dangling pointer is being dereferenced. It also clarifies that the dangling pointer is a result of the variable count, to which the pointer was pointing, going out of scope on line 8. We will use this example through the rest of this paper to illustrate how we generate explanation of the step-by-step execution of a program.

3. MODEL-BASED TUTORS

We have been developing tutors on various programming topics, including expression evaluation [Krishna & Kumar 01, Kumar 05s], selection statements [Kumar & Singhal 00f], loops [Dancik & Kumar 03], pointers in C++ [Kumar 01, Kumar 02f], parameter passing in programming languages [Shah & Kumar 02] and classes [Kostadinov & Kumar 03]. All these tutors target program analysis — evaluating expressions, predicting the output of programs and debugging programs. All of them provide explanation of step-by-step execution of the programs presented in the problems.

In order to increase the repertoire of problems, we have used templates to generate problems in these tutors. We will discuss the details of templates and problem generation in section 3.1. In order to be able to automatically generate explanation of step-by-step execution of programs, we have used principles of Model-Based Reasoning [Davis 84] to model the domain in these tutors [Kumar 02i]. We will present details of domain modeling in section 3.2. In Section 3.3, we will present the architecture of our model-based, template-driven tutors. In Section 3.4, we will present an algorithm to automatically generate explanation of the step-by-step execution of programs. Finally, in Section 3.5, we will present a complete example of a template on the output of a for loop, the corresponding problem, its program model and the explanation generated for it.

3.1 Templates and Problem Generation

Limited problem set is a drawback of encoding a finite number of problems in a tutor [Mitrovic 03, Suraweera & Mitrovic 04]. In order to address this concern, we use problem templates rather than problems in our tutor. This scheme is similar to problem generation in CAPA [Kashy et al 93], CHARLIE [Barker 97] and SAIL [Bridgeman et al 00s].

We use templates to represent programs that serve as the basis of problems, and use a BNF-like notation to represent the templates [Koffman & Perry 76]. In the templates, we use non-terminals for data types, identifiers (variable and function names), and literal constants in the program. We use additional notation to constrain the values of these non-terminals where necessary. The tutor generates the program for a problem by substituting random values for the non-terminals in a template.
E.g., consider the following template on C++ pointers:

```c
{<F0>()<{T1><P1>;;<T1><V1>=<R1 #1<=R1<=10;#>;<P1>=<V1>;}<<<P1>;;}}
```

In the template, <F> refers to a function. <F0> always refers to `main()`. <T1> is a non-terminal for data type. It may be instantiated to any of the data types in C++. <P1> refers to a pointer variable, and <V1> refers to a primitive variable, both of type <T1>. <R1> refers to a random value, constrained to be in the range 1 ... 10. It is assigned to the variable <V1> during initialization.

The following is an example of a program generated from the above template:

```c
1. void main()
2. {
3.     int * ptr;
4.     {
5.         int count = 5;
6.         ptr = &count;
7.     }
8.     cout << *ptr;
9. }
```

Note that <T1> was instantiated to `int` data type; <P1> was translated to the variable `ptr`, <V1> was translated to the variable `count`, and <R1> was instantiated to the value 5.

If a template contains n non-terminals, and the range of possible values for the i\textsuperscript{th} non-terminal is $R_i$, $R_1 \times R_2 \times \ldots \times R_n$ non-identical programs/problems can be generated from the single template. Since the tutor selects values for the non-terminals randomly, the likelihood is miniscule that two problems generated from a template are identical. This enables the tutor to present multiple non-identical instances of a problem, either to the same user on different occasions to provide for repetitive practice, or to different users on the same occasion to prevent plagiarism / cheating. It also makes it possible for the tutor to use the same set of templates for learning (with feedback) and assessment (without feedback). Typically, the tutor contains 200+ templates for each topic. Therefore, the tutor never runs out of “new” problems - the student can practice with the tutor for as long as (s)he pleases.

Substitution of random values for the non-terminals in a template does not alter the semantics of the resulting program, i.e., all the programs generated from a template, though distinct, behave identically. Yet, their outputs may not all be the same, since literal constants are also randomized in the programs. Finally, the Problem Sequencer, which is a part of the traditional Tutor Model, never presents the problems generated from a template back to back. This minimizes the likelihood that the learner can guess the answer to a problem.

Using templates has an added advantage in programming domain: the same templates can be used to generate problems in different programming languages, as long as the languages share the same semantic model. E.g., C++, Java and C# share the same semantic model for most of the imperative programming constructs. Therefore, the same templates can be used for all these three languages as long as the syntax of the appropriate language is used to generate the program code.

### 3.2 Domain Modeling

Researchers have used various technologies to model the domain in an intelligent tutor, including rules (e.g., production rules used in ACT-R theory [Anderson93]), constraints (e.g., [Mitrovic 03, Suraweera & Mitrovic 04]), and cases (e.g., [Reyes & Sison 01]). We use principles of Model-Based Reasoning [Davis 84] to model the domain in our programming tutors [Kumar 02i].

The model of a “device” used in model-based reasoning consists of the structure and behavior of the device. The structure describes the components of the device, and how they are interconnected within
the device. The structure of complex devices is often represented as a hierarchy of devices and components: the components at one level are aggregated to form a device at the next higher level, and this device in turn serves as a component at the subsequent level.

The behavior of a component or device describes the relationship between its inputs and outputs. The behavior may be represented as a mathematical function, procedure, or state diagram. The behaviors of components are local – each component is responsible only for its behavior. The behavior of a device is obtained by composing the behaviors of its components. This modularity of design promotes scalability - new components can be added to a model without affecting the behaviors of other components.

When obtaining the behavior of a device by composing the behaviors of its components, typically, the state diagrams of the components are composed to obtain the state diagram of the device (e.g., [Bouwer & Bredeweg 02, Rickel & Porter97]). This can lead to combinatorial explosion of states in the device. Earlier, we had proposed component-ontological representation [Kumar & Upadhyaya 98][Kumar & Upadhyaya 95], wherein, the behavior of a device is obtained by composing the inputs/outputs of the components rather than their states. The advantages of component-ontological representation are that it is tractable, and promotes better reuse of component models.

The programming domain lends itself well to model-based representation. Programming objects are treated as components, e.g., variables, blocks, functions, etc. In the structural hierarchy, variables are part of a block, blocks are part of a function and functions are part of a program. This hierarchy reflects the scope hierarchy in the program. The structural interconnections among the components are determined entirely based on the flow of data and control during the execution of the program.

We use state diagrams to model the behavior of data components such as variables. The state diagram of the behavior of primitive variables is shown in figure 1. The states of a primitive variable include declared, assigned and de-allocated. Declaration, Assignment, Input, Initialization, Referencing and Out-of-Scope are actions applied to the variable.

![Figure 1: States in the behaviour of a primitive variable](image)

The behavior of the program is determined by the flow of data and control during the execution of the program.

Consider again, the C++ pointer code listed earlier. The structure of the program model generated for this program is shown in figure 2.

![Figure 2: Program model for a C++ program involving pointers](image)
Program is the top level device in the program model. The components inside the program are variables (such as global variables) and children (functions). In this case, the function `main()` is the only component inside Program. The components inside the function `main()` are the pointer variable `ptr` and a block. The variable `count` is a component of the block. The behavior of `count` is modeled by the state diagram shown in figure 1.

One advantage of using model-based reasoning is that the resulting domain model doubles as the expert module. Since the domain model can be simulated to generate the correct answer, we do not have to encode correct answers for problems. This in turn makes it possible to incorporate the following features into the tutor:

- The tutor can extend its repertoire of problems by encoding problem templates instead of actual problems. It can then dynamically generate new problems as random instances of the parameterized templates.
- The tutor can potentially solve problems entered by the student.

In both these cases, the tutor should also be able to generate feedback dynamically. We will propose algorithms for doing so in the next section.

3.3 Architecture of our Model-Based Tutors

The architecture of our model-based programming tutors is shown in figure 3.

- The Problem Sequencer, which is part of the traditional Tutor Model, picks a template from the database of problem templates.
- The Domain Model processes this template to produce the Program Model, e.g., see figure 2.
- The Program Model parses the template to produce the program code, which is part of the problem statement.
- The Program Model executes the template to generate the actual answer and explanation of the step-by-step execution of the program.
- The grader, which is part of the traditional Tutor Model, grades the student’s answer by comparing it with the actual answer. It determines the credit for the answer and saves it in the Student Model.
- The Problem Sequencer consults this updated Student Model to pick the next template from the database of problem templates.

![Figure 3: Architecture of Our Model-Based Tutors](image)

3.4 Explanation Generation
Our tutors provide demand feedback. Demand feedback may contribute to better retention (e.g., [Schooler & Anderson 90]) and transfer of skills [Lee 92]. According to Guidance Hypothesis [Schmidt et al 89], demand feedback promotes the use of evaluative tasks, which include error-detection and correction, the mainstays of program debugging. Therefore, demand feedback is very appropriate for our tutors on program analysis.

The demand feedback provided by our tutor includes the following elements: the actual answer computed by the tutor; the answer entered by the student; the grade for the student's answer; and an explanation of the step-by-step execution of the program.

The tutor generates explanation of the step-by-step execution of a program using simulation and reflection [Neuper 01]. During simulation, the lines of code in the program are executed on the program model, i.e., the components in the program model undergo changes in their state as dictated by the lines of code. The program model narrates these changes, which is an explanation of the behavior of the program. Reflection is the process of examining and explaining one's state. During reflection, the components in the program model examine and explain their state.

We use a two-stage algorithm consisting of simulation and reflection [Kumar 04a]:

- **Simulation / Process Explanation:** This is similar to program tracing. For each line of code, the program model identifies the components participating in the line of code, and explains the 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 the variables. Since the lines of code are executed in the order specified by the program, the resulting feedback narrative is coherent.

- **Reflection / Component Explanation:** For each line of code, the components participating in the line of code generate this explanation. If an action is applied to a component that is not supported by the current state of the component, the component explains why the attempt to apply the action is an error. 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 $S_1 \ ... \ S_i$. For example, consider the states of a primitive variable as shown in figure 1. If a variable is deallocated when an attempt is made to assign to it, which results in an error, the variable steps through and generates explanation for declared, assigned and deallocated states. This explanation may be presented in one of two forms:
  - **Detailed form:** The explanation for all the states $S_1 \ ... \ S_i$ is included in order, culminating in the error that is generated when the action $A_j$ is applied. This describes not only the error but also the sequence of events that lead up to it.
  - **Abbreviated form:** The explanation for only the last state $S_i$ is included, and how it contributes to the error when the action $A_j$ is applied. This describes the error, but not the sequence of events that lead up to it.

Component explanation is inserted into 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, and only the abbreviated form of component explanation is inserted. The algorithm for inserting component explanation within process explanation is as follows:

```plaintext
Execute the code line by line:
For each line of code
    Generate process explanation
    For each component involved in the line
        If the component reached an error state on this line
            Generate abbreviated component explanation for the component
            Insert it after the process explanation for the line
```

The resulting explanation may be presented to the student in one of two forms:

- **Simulative Feedback:** This includes a complete explanation of the behavior of the program. This feedback is used for novices, and for the first few problems in a tutoring session.
Diagnostic Feedback: The explanation includes only the following elements:

- The abbreviated explanation generated by the components. This corresponds to just the description of errors in the program.
- Process explanation corresponding to the inputs to the program and the outputs from it. This feedback is used after the first few problems in a tutoring session, and for advanced students.

The two-stage algorithm can be applied to any discrete domain for generating explanation of step-by-step simulation as long as the following conditions hold:

- 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 the structure of a device contribute to the behavior of the device.

3.4.1 Example

Consider the C++ program on pointers again:

```cpp
1. void main()
2. {
3.   int * ptr;
4.   { int count = 5;
5.     ptr = &count;
6.   } cout << *ptr;
7. }
```

The correct answer to the above problem is:

On line 9, Dangling pointer for ptr: Dereferencing pointer after the variable to which it points goes out of scope

We will now describe how the program model in figure 3 generates an explanation of the step-by-step execution of this program. Unless otherwise mentioned, these lines of explanation are generated during process explanation:

The Program object generates the following lines of explanation when the program starts executing:

When the program is executed:
The program always starts executing from main().

The function main() generates the following preface when it starts executing on line 2:

When the function main() is executed:

The variable ptr generates the following lines of explanation when the pointer is declared on line 4:

Pointer ptr is declared on line 4.

But, it has not been assigned a value yet

The block object generates the following preface when it starts executing on line 5:

When the nested block starting at line 5 is executed:

The variable count generates the following lines of explanation corresponding to its declaration on line 6:

Variable count is declared on line 6.

It is initialized during declaration to 5

The pointer ptr generates the following line of explanation when it is assigned on line 7:

Pointer ptr is assigned to point to count on line 7
The block object generates the following lines of explanation when it exits execution on line 8:
The nested block is exited on line 8
Variable count goes out of scope and is deallocated on line 8

On line 9, the pointer *ptr* attempts to explain a dereferencing operation. Since this operation cannot be applied to the pointer in its current state, the pointer generates the following preface during process explanation:
*An attempt is made to print the value of the variable pointed to by *ptr* on line 9*

Further, pointer *ptr* generates the following lines during component explanation:
*But, variable count has already gone out of the scope of its declaration. Therefore, *ptr* is a dangling pointer.*

Finally, the function *main()* generates the following explanation as it exits execution on line 10:
The function *main()* is exited on line 10
Pointer *ptr* goes out of scope and is deallocated on line 10

The explanation of the step-by-step execution of the program generated by collating the above lines of explanation is shown below:

When the program is executed:
The program always starts executing from *main()*.
When the function *main()* is executed:
Pointer *ptr* 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 count is declared on line 6.
It is initialized during declaration to 5
Pointer *ptr* is assigned to point to count on line 7
The nested block is exited on line 8
Variable count goes out of scope and is deallocated on line 8
*An attempt is made to print the value of the variable pointed to by *ptr* on line 9*
*But, variable count has already gone out of the scope of its declaration. Therefore, *ptr* is a dangling pointer.*
The function *main()* is exited on line 10
Pointer *ptr* goes out of scope and is deallocated on line 10

This explanation is provided as simulative feedback. For diagnostic feedback, only the italicized lines are provided, which are an abbreviated version of the feedback generated during component explanation, along with a preface generated during process explanation:
*An attempt is made to print the value of the variable pointed to by *ptr* on line 9*
*But, variable count has already gone out of the scope of its declaration. Therefore, *ptr* is a dangling pointer.*

The detailed form of the above component explanation is:

At the statement where *ptr* is being dereferenced on line 9:
*ptr* was declared earlier.
*ptr* is currently in scope.
*ptr* was assigned to point to count.
*But, variable count has already gone out of the scope of its declaration. Therefore, *ptr* is a dangling pointer.*
3.5 A Complete Example on for Loops

We will now describe a second example on for loops, the subject of one of the tutors that we have evaluated in this paper. Whereas the student is asked to debug a program in the pointers tutor, the student is asked to predict the output of a program in the for loop tutor, both tasks requiring the student to analyze the given code. We will present a template, the program generated from the template, the program model generated to execute the program and the simulative and diagnostic feedback generated by the tutor for the program.

Consider the following template on for loops in C++:

```c++
{
  <F0>()
  {
    <T1#integer#><V1>;
    <T1><V2>=<R2#integer;1<=R2<=5;#>;
    for(<V1> = <R1#10<=R1<=15;#>;<V1> > <V2>;<V1> = <V1> - 1)
    {
      <<<V1>;
      <<<V2>;
      <V2> = <V2> + 1;
    }
  }
}
```

In the template, <F0> refers to the function main(). <T1> is a non-terminal for a data type, constrained to be an integer data type. <V1> and <V2> are variables of type <T1>. <V2> is initialized during declaration to a value in the range 1 .. 5. In the for loop, <V1> is initialized to a value in the range 10 .. 15. <V1> is compared with <V2> in the condition of the loop and decremented by 1 on each iteration of the loop. Inside the loop body, the values of <V1> and <V2> are printed, and the value of <V2> is incremented by 1.

The following is an example of a program generated from the above template. In the program, <T1> has been instantiated to unsigned data type. <V1> is units and <V2> is rate. rate has been initialized to 3 during declaration. units has been initialized to 11 in the for loop.

```c++
1. // The C++ Program
2. 3. void main()
4. 5. {
6.    unsigned units;
7.    unsigned rate = 3;
8.    for( units = 11;
9.          units > rate;
10.         units = units - 1 )
11.    {
12.      cout << units;
13.      cout << rate;
14.      rate = rate + 1;
15.    }
```

The structure of the program model generated for this program is shown in figure 4.
Figure 4: Program Model of the program with \texttt{for} loop

The correct answer to the above problem is:
11 is printed on line 11
3 is printed on line 12
10 is printed on line 11
4 is printed on line 12
9 is printed on line 11
5 is printed on line 12
8 is printed on line 11
6 is printed on line 12

The program model of figure 4 generates an explanation of the step-by-step execution of this program line by line as described below. Since the program has no bugs, all the lines of explanation are generated during \texttt{process explanation}:

The Program object generates the following preface when the program starts executing:
The program is executed starting from \texttt{main()}.  

The function \texttt{main()} generates the following preface when it starts executing on line 3:
The function \texttt{main()} at line 3 is executed.

The variable \texttt{units} generates the following lines of explanation when it is declared on line 5:
Variable \texttt{units} is declared on line 5.
But, it has not been assigned a value yet.

The variable \texttt{rate} generates the following lines of explanation when it is declared on line 6:
Variable \texttt{rate} is declared on line 6.
It is initialized during declaration to 3

When executing the \texttt{for} loop, \texttt{main()} generates the following preface:
A-for-loop is executed on line 7.

It generates the following explanation of loop initialization:
Initialization: \texttt{units} is assigned 11 on line 7

Thereafter, on each iteration of the loop, \texttt{main()} generates explanation for the evaluation of the condition of the loop, e.g.,
Condition: The for loop condition i.e., \texttt{units} > \texttt{rate} is evaluated on line 8.
It returns \texttt{true}, therefore the for loop is entered on line 10.
If the condition evaluates to false, it generates the following explanation:
Condition: The for loop condition i.e., \texttt{units} > \texttt{rate} is re-evaluated on line 8.
It returns \texttt{false}, therefore the for loop is terminated.

On each iteration of the loop, within the body of the loop, \texttt{units} and \texttt{rate} variables generate explanation for printing their values. Variable \texttt{rate} also generates explanation about its value being incremented, e.g.,
The value of units, i.e., 11 is printed on line 11
The value of rate, i.e., 3 is printed on line 12
rate is incremented to 4 on line 13

On each iteration of the loop, main() generates explanation for the update of the for loop, e.g.,
Update: units is decremented to 10 on line 9

Upon exiting the loop, main() generates the following explanation:
The for loop is exited on line 14.
The remaining code continues execution from line 15.

Finally, the function main() generates the following explanation when it exits execution on line 15:
The function main() is exited on line 15
Variable units goes out of scope and is deallocated on line 15
Variable rate goes out of scope and is deallocated on line 15
This completes the execution of the program.

The explanation of the step-by-step execution of the program generated by collating the above segments
is shown below:

The program is executed starting from main().
The function main() at line 3 is executed.
Variable units is declared on line 5.
But, it has not been assigned a value yet.
Variable rate is declared online 6.
It is initialized during declaration to 3
A for-loop is executed on line 7.
Initialization: units is assigned 11 on line 7
Condition: The for loop condition i.e., units > rate is evaluated on line 8.
   It returns true, therefore the for loop is entered on line 10.
Action: The value of units, i.e., 11 is printed on line 11
The value of rate, i.e., 3 is printed on line 12
rate is incremented to 4 on line 13
Update: units is decremented to 10 on line 9

Condition: The for loop condition i.e., units > rate is re-evaluated on line 8.
   It returns true, therefore the for loop body continues execution.
Action: The value of units, i.e., 10 is printed on line 11
The value of rate, i.e., 4 is printed on line 12
rate is incremented to 5 on line 13
Update: units is decremented to 9 on line 9

Condition: The for loop condition i.e., units > rate is re-evaluated on line 8.
   It returns true, therefore the for loop body continues execution.
Action: The value of units, i.e., 9 is printed on line 11
The value of rate, i.e., 5 is printed on line 12
rate is incremented to 6 on line 13
Update: units is decremented to 8 on line 9

Condition: The for loop condition i.e., units > rate is re-evaluated on line 8.
   It returns true, therefore the for loop body continues execution.
Action: The value of units, i.e., 8 is printed on line 11
The value of rate, i.e., 6 is printed on line 12
rate is incremented to 7 on line 13
Update: units is decremented to 7 on line 9

Condition: The for loop condition i.e., units > rate is re-evaluated on line 8.
It returns false, therefore the for loop is terminated.
The for loop is exited on line 14.
The remaining code continues execution from line 15.

The function main() is exited on line 15
Variable units goes out of scope and is deallocated on line 15
Variable rate goes out of scope and is deallocated on line 15
This completes the execution of the program.

The above explanation constitutes the simulative feedback for the problem. Diagnostic feedback for the above program is:

The value of units, i.e., 11 is printed on line 11
The value of rate, i.e., 3 is printed on line 12
The value of units, i.e., 10 is printed on line 11
The value of rate, i.e., 4 is printed on line 12
The value of units, i.e., 9 is printed on line 11
The value of rate, i.e., 5 is printed on line 12
The value of units, i.e., 8 is printed on line 11
The value of rate, i.e., 6 is printed on line 12

3.6 Model-Based Tutors for Program Analysis

To date, we have developed and evaluated model-based tutors on the following topics:

- Expression evaluation in C++ [Krishna & Kumar 01, Kumar 05s] – the learner is asked to solve
  arithmetic and relational expressions; the tutor provides graphical and textual feedback to the learner;
- for loops in C++ [Dancik & Kumar 03] and selection statements in C++ [Kumar & Singhal 00] – the
  learner is asked to predict the output of the code; the tutor explains the correct execution of the code;
- Encapsulation in C++ classes [Kostadinov & Kumar 03] – the learner is asked to identify syntax and
  semantic errors in the code; the tutor explains the bugs in the code;
- Pointers in C++ [Kumar 01, Kumar 02] – the learner is asked to identify syntactic, semantic and run-
  time errors in the code; the tutor explains the bugs in the code;
- Parameter passing in programming languages [Shah & Kumar 02] – the learner is asked to indicate
  the behavior of a program given a parameter passing mechanism (value, result, value-result, reference or name); the tutor explains the behavior of the code;
- Scope concepts [Kumar 00, Kumar, Schottenfeld & Obringer 00, Fernandes & Kumar 04] – the
  learner is asked to answer questions based on static and dynamic scope, and concepts from their
  implementation [Fernandes & Kumar 05]. The tutor explains the correct answer and why the student’s
  answer is incorrect.

We have used these tutors to supplement classroom instruction and complement the traditional
programming projects assigned in Computer Science courses.

In the next two sections, we will describe two of the tutors – a tutor on parameter passing and a tutor on
for loops. For each tutor, we will briefly describe the domain, and elaborate on the explanation of step-
by-step execution provided by the tutor. We will present details of evaluating the tutor, including the
evaluation protocol and analysis of the collected data.

4. TUTOR ON PARAMETER PASSING
Parameter passing mechanisms are an important part of the study of programming languages. In a typical Comparative Programming Languages course, five different parameter passing mechanisms are discussed: value, result, value-result, reference and name. At least a few of these mechanisms would be new to every student, since no programming language includes all five parameter passing mechanisms. We developed a model-based tutor to help students learn the five parameter passing mechanisms by solving problems.

In each problem, the tutor presents a program consisting of a calling function, a called function, and optional global variables and arrays. It specifies a parameter passing mechanism and asks the student to specify the values at the end of the execution of the program, of the variables used as actual parameters in the function call and any global variables.

The tutor organizes the explanation of the step-by-step execution into four stages: before the function call, during the function call, during the function execution and during the function return.

- **Before the function call:** In the case of value, value-result, name and reference mechanisms, the tutor lists the initial values of the variables used as actual parameters. In the case of result, this is done only if the actual parameters are evaluated at the time of the function call.

- **During the function call:** In the case of value and value-result mechanisms, the tutor identifies the formal parameters into which values of the actual parameters are copied. In the case of reference mechanism, the tutor highlights the aliasing between the actual and formal parameters.

- **During the function execution:** The tutor explains the execution of the called function line by line, indicating at each step, values of any variables that are changed in that step. In the case of parameter passing by name though, the tutor first rewrites the body of the called function, replacing each instance of a formal parameter by its corresponding actual parameter, before executing the body.

- **During the function return:** In the case of value-result, the tutor indicates how the values of the formal parameters are copied back into the corresponding actual parameters. In the case of result, the tutor tailors its feedback based on whether the actual parameters are evaluated at the time of the function call or function return. In both the parameter passing mechanisms, it takes into account whether the actual parameters are evaluated left to right or right to left, especially when a variable is used as an actual parameter more than once in a function call.

### 4.1 Example

We will present a typical problem and the explanation of step-by-step execution provided by the tutor. The problem includes a program and a problem statement. The following is a typical program presented by the tutor.

```c
1. double area = 1;
2. double depth[8] = {1,1,1,1,1,1,1,1};
3. 
4. void main()
5. {
6.     convert( area, depth[area] );
7. }
8. 
9. void convert(double length, double height)
10. {
11.     length = 3;
12.     height = depth[area] + 3;
13.     area = area + 3;
14.     depth[area] = depth[length] + 3;
15. }
```
The associated problem statement is: Assume that all the parameters are passed by Value-Result when main() calls the function convert(). Indicate the final value of the variable area and all the changes to the elements of the array depth after the execution of the program.

The correct answer for the problem is:
depth[8] = {1,4,1,4,1,1,1,1}.
area = 3.

The explanation of the step-by-step execution of the program provided to the student follows:

Parameters are passed by Value Result:
When main() calls the function convert(), the values of actual parameters are:
area = 1
depth[1] = 1

During the call to the function convert():
Since parameters are passed by Value Result, the value of each actual parameter is copied into the corresponding formal parameter.
The value of area, i.e., 1 is copied into length.
The value of depth[1], i.e., 1 is copied into height.

During the execution of the function convert():
On line 11, the value of length is changed to 3.
On line 12, the value of height is changed to 4.
On line 13, the value of area is changed to 4.
On line 14, the value of depth[4] is changed to 4.

During the return from the function convert():
Since parameters are passed by Value Result, the values of formal parameters are copied back into the corresponding actual parameters.
Since actual parameters are evaluated left to right,
The value of length, i.e., 3 is copied into area
The value of height, i.e., 4 is copied into depth[1]
A snapshot of the tutor is shown in figure 5. The program is presented in the top left panel. The problem statement is presented in the right panel. After entering the answer, the student clicks on the “Check My Answer” button at the bottom of the right panel. The feedback, including explanation of the step-by-step execution of the program is presented in the bottom left panel. After reading the feedback, the student clicks on the “Create New Problem” button in the center of the left panel. Therefore, the student proceeds in a clock-wise fashion around the tutor window for every problem.

4.2 Evaluation Protocol

We evaluated the tutor in spring and fall 2002 in our junior/senior level Comparative Programming Languages course [Shah & Kumar 02]. Our hypothesis was that the explanation of step-by-step execution of the program helps students learn program analysis.

Protocol: We used a within-subjects design, i.e., the same students served as both control and experimental groups, although on different sections of the tutor. We evaluated the tutor in four consecutive stages:

- Tryout Stage: The students solved problems on parameter passing by reference, a parameter passing mechanism that was very familiar to all the students. This stage, which lasted 5 minutes, helped familiarize students with the user interface of the tutor.
• **Practice Effect Stage:** Students answered a written pre-test for 8 minutes, followed by a written post-test for 8 minutes, both on parameter passing by result. They did not break (for any practice) in between the two tests. We used this stage to assess practice effect, i.e., any improvement from pre-test to post-test simply by virtue of increased familiarity of the students with the types of problems presented in the tests.

• **Minimal Feedback Stage:** We used this stage to evaluate the effectiveness of a tutor that provided no feedback except whether the user’s answer was correct or wrong. For this stage, students answered problems on parameter passing by value-result. The students first answered a written pre-test for 8 minutes, practiced with the tutor for 12 minutes, and answered a written post-test for 8 minutes.

• **Detailed Feedback Stage:** We used this stage to evaluate the effectiveness of a tutor that provided simulative feedback, i.e., feedback that included explanation of the step-by-step execution of the program. For this stage, students answered problems on parameter passing by name, arguably the mechanism with which they were the least familiar. Once again, the students first answered a written pre-test for 8 minutes, practiced with the tutor for 12 minutes, and answered a written post-test for 8 minutes.

Each pre-test and post-test contained 8 problems, and each problem was worth 4-5 points, for a total raw score of 32-35 points on the test. We produced the pre-tests and post-tests by typesetting problems that were also generated by the tutor. We ordered the problems so that the sequences of problems on the pre-test and the post-test were comparable. We graded the tests manually. The tutor used for practice was configured to generate problems ad-infinitum. The students did not have access to the tutor before the practice session. They did not have access to their textbook, notes, or any other reference material between the pre-test and the post-test. Therefore, any change of score from the pre-test to the post-test was attributable solely to the tutor used for practice.

4.3 Evaluation Results

In table 1, we have listed the average and standard deviation of the pre-test scores, post-test scores and the pre-post change in score for the three stages of evaluation. We have also noted the maximum possible score for each test and listed the t-test 2-tailed $p$-value of the difference between pre-test and post-test scores. Clearly, there was a statistically significant improvement from pretest to post-test when students received simulative feedback during the interim practice. The change from pretest to post-test was small and not statistically significant when students received minimal feedback. The most curious result is for the stage when students received no practice between the pretest and the post-test: the average score improved by more than one pre-test standard deviation, and the improvement was statistically significant.

<table>
<thead>
<tr>
<th>Score</th>
<th>Pre-Test</th>
<th>Post-Test</th>
<th>Change</th>
<th>$p$-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Practice (N=15, Max=34)</td>
<td>Average</td>
<td>10.40</td>
<td>15.87</td>
<td>5.47</td>
</tr>
<tr>
<td></td>
<td>Std-Dev</td>
<td>4.79</td>
<td>7.64</td>
<td>5.29</td>
</tr>
<tr>
<td>Minimal Feedback (N=15, Max=32)</td>
<td>Average</td>
<td>14.27</td>
<td>16.33</td>
<td>2.07</td>
</tr>
<tr>
<td></td>
<td>Std-Dev</td>
<td>7.69</td>
<td>7.60</td>
<td>6.56</td>
</tr>
<tr>
<td>Simulative Feedback (N=14, Max=33)</td>
<td>Average</td>
<td>12.21</td>
<td>22.64</td>
<td>10.43</td>
</tr>
<tr>
<td></td>
<td>Std-Dev</td>
<td>7.84</td>
<td>8.40</td>
<td>8.34</td>
</tr>
</tbody>
</table>

**Table 1:** Evaluation of Parameter Passing Tutor in 2002 - Scores

Upon closer examination, we found that there was also an increase in the number of problems solved by students from pretest to post-test in all three evaluation stages, as shown in table 2. This increase was statistically significant in the first two stages, and not statistically significant in the last stage. This suggests practice effect: students grew more familiar with the interface of the tutor, and were hence able to attempt more problems as they progressed from stage to stage and from each pretest to post-test.

<table>
<thead>
<tr>
<th>Problems</th>
<th>Pre-Test</th>
<th>Post-Test</th>
<th>Change</th>
<th>$p$-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Practice (N=15, Max=8)</td>
<td>Average</td>
<td>3.67</td>
<td>5.13</td>
<td>1.47</td>
</tr>
<tr>
<td></td>
<td>Std-Dev</td>
<td>1.59</td>
<td>1.64</td>
<td>1.36</td>
</tr>
<tr>
<td>Minimal Feedback</td>
<td>Average</td>
<td>4.87</td>
<td>5.33</td>
<td>0.47</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Score</th>
<th>Pre-Test</th>
<th>Post-Test</th>
<th>Change</th>
<th>$p$-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Practice (N=15, Max=34)</td>
<td>Average</td>
<td>10.40</td>
<td>15.87</td>
<td>5.47</td>
</tr>
<tr>
<td></td>
<td>Std-Dev</td>
<td>4.79</td>
<td>7.64</td>
<td>5.29</td>
</tr>
<tr>
<td>Minimal Feedback (N=15, Max=32)</td>
<td>Average</td>
<td>14.27</td>
<td>16.33</td>
<td>2.07</td>
</tr>
<tr>
<td></td>
<td>Std-Dev</td>
<td>7.69</td>
<td>7.60</td>
<td>6.56</td>
</tr>
<tr>
<td>Simulative Feedback (N=14, Max=33)</td>
<td>Average</td>
<td>12.21</td>
<td>22.64</td>
<td>10.43</td>
</tr>
<tr>
<td></td>
<td>Std-Dev</td>
<td>7.84</td>
<td>8.40</td>
<td>8.34</td>
</tr>
</tbody>
</table>
In order to eliminate this practice effect, we considered the score per problem instead of the raw score. We have listed the score per problem in table 3, along with the maximum score per problem that represents the ceiling effect for each test. Note that the score per problem did not increase from pretest to post-test during the first two stages, although there was ample room for it to increase. But, it did increase by one pre-test standard deviation, and the increase was statistically significant (2-tailed p < 0.05) when simulative feedback was provided. This supports our hypothesis that the explanation of step-by-step execution of programs helps students learn program analysis.

### Table 2: Evaluation of Parameter Passing Tutor in 2002- Problems Solved

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Std-Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Practice</td>
<td>1.81</td>
<td>1.80</td>
</tr>
<tr>
<td>Minimal Feedback</td>
<td>5.50</td>
<td>6.57</td>
</tr>
<tr>
<td>Simulative Feedback</td>
<td>2.10</td>
<td>1.60</td>
</tr>
</tbody>
</table>

Table 3: Evaluation of Parameter Passing Tutor in 2002- Score Per Problem

We performed a 3 X 2 repeated measures ANOVA with the type of feedback (none, minimal or simulative) as the first factor and pretest-post-test as the second factor. Our findings were:

- There was no significant effect of the type of feedback \[ F(2, 26) = 0.63, p=0.55 \].
- There was a significant pretest-post-test effect \[ F(1, 26)=5.90, p=0.03 \].
- There was a significant interaction between feedback type and pretest-post-test \[ F(2,26)=6.80, p=0.01 \] (See figure 6). The results of three paired t-tests (See table 3) revealed a significant pretest-post-test difference for simulative feedback only \[ t(13)=4.077, p=0.001 \]. These results suggest that no practice and minimal feedback have no effect on pretest-post-test differences.

Parameter passing by Result (used with no practice), Value Result (used with minimal feedback) and Name (used with simulative feedback) are sufficiently different from each other that we did not expect students' work with one parameter passing mechanism to influence their performance on the next mechanism. Moreover, such influence, if any would manifest itself in the pre-test just as much as the post-test for the next mechanism. Since, we considered the change from pre-test to post-test in our analysis, this influence, if any, could not affect our results.
5. TUTOR ON for LOOPS

Counter-controlled loops, i.e., for loops provide a powerful mechanism for repetition in programming languages. An understanding of this topic is expected at the level of introductory computer science, and mastery of this topic is essential for computer science students who wish to advance to higher-level programming concepts. Understanding for loops requires comprehension of all the components of the loop – initialization, condition, update, and body – which may or may not always be present in a loop. Counter-controlled loops can be classified as up-counting or down-counting, depending on the update statement. In our experience, students find down-counting loops more difficult than up-counting loops. We developed a model-based tutor for for loops to help students learn all these concepts by solving problems.

In each problem, the tutor presents a C++ program containing a for loop. The student is asked to predict the output of the loop and the number of times it iterates. The tutor presents the explanation of the step-by-step execution of the program organized into three stages:

- **Pre-loop:** The tutor describes any variable declarations, assignments, or output statements that are executed before entering the for loop.
- **During the loop execution:** The tutor describes the execution of the loop under the following four titles, which correspond to the four components of the loop:
  - **Initialization:** The tutor explains how the loop counter is initialized, either with a literal constant or a variable declared earlier.
  - **Condition:** The tutor evaluates the condition of the loop, and based on the result of evaluating the condition, indicates whether the loop is entered, exited, or not entered at all.
- **Loop Body/Action:** The tutor explains the execution of the statements within the body of the loop, such as output statements and variable assignments.
- **Update:** The tutor explains how the loop counter is updated and lists the new value of the counter.

- **Post-loop:** The tutor explains the execution of any statements that follow the loop.

The tutor explains the pre-loop and initialization segments only once in the beginning. It explains the condition, loop body and update segments for every iteration of the loop. It explains the post-loop segment once after the loop is exited. Please see Section 3.5 for an example of the explanation of step-by-step execution generated by the tutor.

A snapshot of the tutor is shown in Figure 7. The program is presented in the top left panel. The problem statement and controls to input student's answers are presented in the top right panel. After entering the answer, the student clicks on the “Check My Answer” button in the middle of the right panel (not shown in the figure since it is displayed only until the student submits the answer, and is hidden thereafter). The feedback, including explanation of the step-by-step execution of the program is presented in the bottom right panel. After reading the feedback, the student clicks on the “Next Problem” button at the bottom of the right panel. Therefore, the student proceeds in a clock-wise fashion around the tutor window for every problem.

![Figure 7: Snapshot of the tutor on for loop](image)

### 5.1 Evaluation Protocol
We evaluated the feedback provided by the for loop tutor in spring 2003 in our introductory Computer Science course [Dancik & Kumar 03]. Our hypothesis was that the explanation of step-by-step execution of the program helps students learn program analysis.

Protocol: We evaluated the tutor in two stages – the first for up-counting loops and the second for down-counting loops. We used a partial crossover design – we randomly divided the class into two groups. Each group served as the control group in one stage and as the test group in the other. We used the pre-test-practice-post-test sequence in each stage:

- **Up-counting stage:**
  - Pre-test: Both groups answered a pre-test for 5 minutes.
  - Practice: Both groups spent 8 minutes practicing problem-solving. The control group used a printed workbook for practice. The workbook contained practice problems and answers, but no explanation. The test group used the tutor, configured to provide simulative feedback.
  - Post-test: Both groups answered a post-test for 5 minutes.
- **Down-counting stage:** We switched the control and test groups and repeated the protocol used for the up-counting stage.

For the pre-test and post-test, both groups used the tutor, configured to provide no feedback. During the tests, the tutor was configured to generate problems ad-infinitum. The tutor graded the student answers as correct or wrong, without any partial credit. We produced the printed workbooks used by the control group for practice by typesetting 50 problems that were also generated by the tutor. The tutor generated these problems using the same sequence of templates that it used for pretest, practice, and post-test. We included answers for all the problems at the back of the workbook, first for the odd-numbered questions, and then for the even-numbered ones. But, we did not include any explanation of the correct answers. We deemed 50 problems were sufficient to provide practice for 8 minutes.

The students did not have access to the tutor before the practice session. They did not have access to their textbook, notes, or any other reference material between the pretest and the post-test. Therefore, any change in score from the pretest to the post-test was attributable solely to the use of the tutor or the printed workbook during practice.

5.2 Evaluation Results

We have listed the average and standard deviation of the pre-test scores, post-test scores and pre-post change in scores for workbook and tutor users on up-counting loops in table 4, and on down-counting loops in table 5. We have listed the t-test 2-tailed p-value of the difference between pre-test and post-test scores of workbook and tutor users, and between the workbook and tutors users on the pre-test and the post-test. Finally, we have listed the effect size, calculated as the difference between the means of workbook and tutor users on the post-test, divided by the standard deviation of the workbook users on the post-test. Inconsistency in n-values across the tables is a result of some students not attempting any problem of a given type. In the tables, note that the changes from pre-test to post-test are statistically significant for both workbook and tutor users, but the difference between workbook and tutor users is not statistically significant on the pre-test or the post-test.

<table>
<thead>
<tr>
<th>Up-Counting Loops: Scores</th>
<th>Pre-Test</th>
<th>Post-Test</th>
<th>Change</th>
<th>p-value</th>
<th>Effect Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Workbook Users (N=25)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>1.44</td>
<td>2.76</td>
<td>1.32</td>
<td>0.0012</td>
<td>0.47</td>
</tr>
<tr>
<td>Std-Dev</td>
<td>1.69</td>
<td>2.11</td>
<td>1.80</td>
<td>0.0053</td>
<td>0.25</td>
</tr>
<tr>
<td>Tutor Users (N=24)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average</td>
<td>0.75</td>
<td>3.75</td>
<td>3.00</td>
<td>0.00002</td>
<td></td>
</tr>
<tr>
<td>Std-Dev</td>
<td>1.45</td>
<td>2.97</td>
<td>2.73</td>
<td></td>
<td></td>
</tr>
<tr>
<td>p-value</td>
<td>0.13</td>
<td>0.18</td>
<td>0.014</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4: Up-counting loops (Spring 2003) – Scores of Workbook and Tutor users
Table 5: Down-counting loops (Spring 2003) – Scores of Workbook and Tutor users

<table>
<thead>
<tr>
<th></th>
<th>Average</th>
<th>Std-Dev</th>
<th>p-value</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Tutor Users (N=22)</strong></td>
<td>2.14</td>
<td>6.27</td>
<td>4.14</td>
<td>0.00001</td>
</tr>
<tr>
<td><strong>Std-Dev</strong></td>
<td>2.46</td>
<td>4.28</td>
<td>3.41</td>
<td>0.011</td>
</tr>
<tr>
<td><strong>p-value</strong></td>
<td>0.07</td>
<td>0.41</td>
<td>0.011</td>
<td>0.011</td>
</tr>
</tbody>
</table>

Table 6: Up-counting loops (Spring 2003) – Number of Problems Solved by Workbook and Tutor users

<table>
<thead>
<tr>
<th></th>
<th>Pre-Test</th>
<th>Post-Test</th>
<th>Change</th>
<th>p-value</th>
<th>Effect Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Workbook Users (N=25)</strong></td>
<td>4.60</td>
<td>7.96</td>
<td>3.36</td>
<td>0.00003</td>
<td>0.009</td>
</tr>
<tr>
<td><strong>Std-Dev</strong></td>
<td>2.89</td>
<td>4.47</td>
<td>3.24</td>
<td>0.009</td>
<td></td>
</tr>
<tr>
<td><strong>p-value</strong></td>
<td>0.43</td>
<td>0.97</td>
<td>0.50</td>
<td>0.009</td>
<td></td>
</tr>
</tbody>
</table>

Table 7: Down-counting loops (Spring 2003) – Number of Problems Solved by Workbook and Tutor users

<table>
<thead>
<tr>
<th></th>
<th>Pre-Test</th>
<th>Post-Test</th>
<th>Change</th>
<th>p-value</th>
<th>Effect Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Workbook Users (N=25)</strong></td>
<td>7.12</td>
<td>9.36</td>
<td>2.24</td>
<td>0.00014</td>
<td>0.011</td>
</tr>
<tr>
<td><strong>Std-Dev</strong></td>
<td>3.26</td>
<td>4.54</td>
<td>2.47</td>
<td>0.011</td>
<td></td>
</tr>
<tr>
<td><strong>p-value</strong></td>
<td>0.42</td>
<td>0.97</td>
<td>0.27</td>
<td>0.011</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Up-counting loops (Spring 2003) – Score Per Problem of Workbook and Tutor Users

<table>
<thead>
<tr>
<th></th>
<th>Pre-Test</th>
<th>Post-Test</th>
<th>Change</th>
<th>p-value</th>
<th>Effect Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Workbook Users (N=25)</strong></td>
<td>0.24</td>
<td>0.38</td>
<td>0.13</td>
<td>0.0838</td>
<td>0.15</td>
</tr>
<tr>
<td><strong>Std-Dev</strong></td>
<td>0.26</td>
<td>0.27</td>
<td>0.36</td>
<td>0.15</td>
<td></td>
</tr>
<tr>
<td><strong>p-value</strong></td>
<td>0.03</td>
<td>0.62</td>
<td>0.06</td>
<td>0.15</td>
<td></td>
</tr>
</tbody>
</table>

Table 9: Down-counting loops (Spring 2003) – Score Per Problem of Workbook and Tutor Users

<table>
<thead>
<tr>
<th></th>
<th>Pre-Test</th>
<th>Post-Test</th>
<th>Change</th>
<th>p-value</th>
<th>Effect Size</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Workbook Users (N=25)</strong></td>
<td>0.42</td>
<td>0.48</td>
<td>0.05</td>
<td>0.1883</td>
<td>0.41</td>
</tr>
<tr>
<td><strong>Std-Dev</strong></td>
<td>0.29</td>
<td>0.32</td>
<td>0.20</td>
<td>0.41</td>
<td></td>
</tr>
<tr>
<td><strong>p-value</strong></td>
<td>0.06</td>
<td>0.16</td>
<td>0.007</td>
<td>0.41</td>
<td></td>
</tr>
</tbody>
</table>

Clearly, the improvement in the percentage of correctly solved problems from the pre-test to the post-test was greater for tutor users than for workbook users, although there was ample room for improvement for both groups. The improvement was always statistically significant for tutor users (at \( p < 0.05 \) level), and never so for workbook users. Within each group, there was greater improvement with the tutor than with
the workbook, e.g., a pre-post improvement of 0.13 with workbook versus 0.33 with tutor for one group, and an improvement of 0.05 with workbook versus 0.31 with tutor for the other group.

Consider the effect sizes. There was a larger effect size for up-counting loops than down-counting loops (0.47 versus 0.25) when raw scores were analyzed (tables 4 and 5) and vice versa ((0.15 versus 0.41) when score per problem was considered (tables 8 and 9). This once again highlights the influence of practice effect on raw scores.

We conducted a 2X2 repeated measures ANOVA on the percentage of problems solved correctly, with the treatment (workbook versus tutor) and topic (up-counting versus down-counting loops) as between-subjects factors and pretest-post-test as repeated measures. Our findings were:

- There was a significant main effect for pre-test versus post-test \[F(1,92) = 46.073, p < 0.001\] - post-test scored significantly higher than pre-test.
- There was a significant interaction between the treatment (workbook versus tutor) and time repeated measure \[F(1,92) = 14.083, p < 0.001\]: while workbook use showed a modest increase, tutor use showed a much more significant increase (See figure 8).
- There was no significant interaction between the topic (up-counting versus down-counting) and time repeated measure \[F(1,92) = 0.197, p = 0.658\].
- There was no significant 3-way interaction between pre-test-post-test, treatment (workbook versus tutor) and topic (up-counting versus down-counting) \[F(1,92) = 0.678, p = 0.413\].

![Workbook Vs Tutor from Pre-test to Post-test](image)

Figure 8: For Loops (Spring 2003) - Interaction between the treatment (Workbook Vs Tutor) and time repeated measure

<table>
<thead>
<tr>
<th>Problems</th>
<th>Workbook Users</th>
<th>Tutor Users</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Up-Counting Loops</td>
<td>N 25</td>
<td>24</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>Average 7.60</td>
<td>7.25</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Std-Dev 4.90</td>
<td>3.45</td>
<td></td>
</tr>
<tr>
<td>Down-Counting Loops</td>
<td>N 25</td>
<td>22</td>
<td>0.70</td>
</tr>
<tr>
<td></td>
<td>Average 9.72</td>
<td>9.14</td>
<td></td>
</tr>
</tbody>
</table>
We have listed the number of problems solved by the students during practice in table 10. Note that there was no statistically significant difference between workbook users and tutor users.

The students worked with up-counting loop problems before down-counting loop problems. So, their work with up-counting loop problems could be expected to influence their performance on down-counting loop problems. However, such influence, if any, would manifest itself in the pre-test just as much as the post-test for the down-counting loop problems. Since, we considered the change from pre-test to post-test in our analysis, this influence, if any, could not affect our results.

Both groups used the tutor for pre-tests and post-tests. Therefore, the difference in the familiarity with the interface of the tutor between the control (workbook users during practice) and experimental (tutor users during practice) groups, if any, is negligible and did not affect our analysis.

6. DISCUSSION

The results of these two evaluations support the hypothesis that the explanation of step-by-step execution of a program helps students learn to analyze programs [Kumar 03b]. In the two evaluations, we compared the explanation of step-by-step execution against not providing any explanation other than whether the answer was correct or wrong (parameter passing tutor only) and listing the correct answer. Could some sort of canned explanation other than/short of an explanation of step-by-step execution also have produced the same improvement in learning? Intuitively, it seems that explanation of the step-by-step execution would be the most beneficial for learning about program analysis. All the same, this question is worth investigating and could be the focus of a future study. Also, we have assumed that the mode of delivery of feedback, i.e., offline in the workbook versus online in the tutor - did not affect the results of our evaluation.

To date, we have implemented and evaluated tutors on arithmetic and relational expressions [Krishna & Kumar 01, Kumar 05s], if/else selection statements [Kumar & Singhal 00f], loops [Dancik & Kumar 03], pointers in C++ [Kumar 01, Kumar 02f], parameter passing in programming languages [Shah & Kumar 02], classes [Kostadinov & Kumar 03] and static and dynamic scope concepts [Kumar 00, Kumar, Schottenfeld & Obringer 00, Fernandes & Kumar 04] and their implementation [Fernandes & Kumar 05]. All except the scope tutors provide explanation of the step-by-step execution of programs. We plan to develop additional tutors on Boolean operators, bitwise operators, assignment operators, multi-way selection statement, functions, arrays and structures. We also plan to extend the tutors to Java and C#.

Research indicates that to-the-point feedback is better than verbose feedback for promoting learning [Winkels 92]. Evidence suggests that advanced students do better at inference and problem solving after reading text if the text is sparse, requiring significant active processing of its meaning; and in contrast, do more poorly when the passage is much more complete, requiring less processing of its deep meaning [McNamara et al 96]. We plan to gradually decrease the amount of detail in the explanation of step-by-step execution in our tutors. Finally, learning is most effective when it is adapted to the needs of the learner [Bloom 84]. Adaptation speeds up learning [Arroyo et al 01, Wainer 90]. We are currently working on adapting the explanation of step-by-step execution to the needs of the learner. We are also working on a suitable interface to present the adaptive explanation.

ACKNOWLEDGMENTS

Partial support for this work was provided by the National Science Foundation’s Course, Curriculum and Laboratory Improvement Program under grant DUE-0088864 and the Educational Innovation Program under grant CNS-0426021. Garrett Dancik implemented the tutor on for loops, and Harsh Shah implemented the tutor on parameter passing. The author would like to thank the anonymous referees and
Tomek Loboda for their constructive feedback. The author is also indebted to Joseph Cataliotti and Peter Rutigliano for their invaluable help with analyzing the results of evaluation.

7. REFERENCES


[Kumar 04a] Kumar, A.N. Generation of Demand Feedback in Intelligent Tutors for Programming. Advances in Artificial Intelligence, Ahmed Tawfik and Scott Goodwin (eds.), Proceedings of The


