

Constructing Contracts: Making Discrete Mathematics Relevant to Beginning Programmers

TIMOTHY S. GEGG-HARRISON

Winona State University

Although computer scientists understand the importance of discrete mathematics to the foundations of their field, computer science (CS) students do not always see the relevance. Thus, it is important to find a way to show students its relevance. The concept of program correctness is generally taught as an activity independent of the programming process, hence many CS students perceive it as unnecessary, and even irrelevant. The concept of contracts, on the other hand, is generally taught as an integral part of the programming process. Most CS students have little difficulty understanding the need to establish contracts via preconditions and postconditions. In order to improve teaching program correctness concepts, we implemented ProVIDE, an enhanced integrated development environment (IDE). ProVIDE assists student programmers in contract construction. Rather than asking for both a precondition and postcondition for each of the student's methods, ProVIDE asks the student to simply supply a postcondition. ProVIDE then helps the student construct the appropriate precondition by leading him or her through an axiomatic proof of the method's correctness. Thus, the proof of the method's correctness is a side-effect of the student's need to construct an appropriate precondition.

Categories and Subject Descriptors: D.2.4 [Software Engineering]: Software/Program Verification – *Programming by contract; Correctness proofs; Validation; Formal methods*; F.3.1 [Logics and Meanings of Programs]: Specifying, Verifying, and Reasoning about Programs – *Assertions; Invariants; Pre- and post-conditions; Specification techniques*; G.2.3 [Discrete Mathematics]: Applications; K.3.2 [Computers and Education]: Computer and Information Science Education – *Computer science education*

General Terms: Documentation, Verification

Additional Key Words and Phrases: Formal methods, programming by contract, axiomatic semantics, program verification, loop invariants, weakest preconditions, programming tools.

1. INTRODUCTION

Component-based software construction has become one of more successful methodologies for producing quality software on schedule. It is no longer possible for a single programmer to create a commercial-grade application alone. In an attempt to ease the software development effort, component-based software construction capitalizes on the success of “plug and play” in the hardware industry and carries it over to the software industry in the form of software components. By breaking down the overall functionality of a software system into independent, replaceable software components, component-based software construction provides an ideal methodology for separating coding responsibilities among a group of independent programmers. Communication between the programmers is maintained by software contracts that precisely define the functionality of the components that they are responsible for creating. Implementation

This research was supported in part by National Science Foundation grant DUE-0127483/0353132.

Author's address: T. Gegg-Harrison, Dept. of Computer Science, Winona State University, Winona, MN 55987; email: tgegggharrison@winona.edu; WWW: <http://cs.winona.edu/tgegggharrison>.

Permission to make digital/hard copy of part of this work for personal or classroom use is granted without fee provided that the copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication, and its date of appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Permission may be requested from the Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036, USA, fax: +1 (212) 869-0481, permission@acm.org

© 2006 ACM 1531-4278/06/0300-ART3 \$5.00

details are irrelevant as long as the programmers abide by the contracts. There are four major types of contract [Beugnard et al. 1999]:

- (1) syntactic contracts
- (2) behavioral contracts
- (3) synchronization contracts
- (4) quality of service contracts

Syntactic contracts consist of the basic “signature” of the method (i.e., the name of the method, its return type, the type and number of parameters that it expects, and exceptions that it may throw). Behavioral contracts consist of method preconditions, method postconditions, and class invariants. Although important in complex object-oriented systems, synchronization contracts (which identify the global behavior of objects in terms of synchronizations between method calls) and quality of service contracts (which quantify the expected behavior of the method in terms of response times and the quality of the result) are generally beyond the scope of beginning programmers.

From the very beginning, undergraduate computer science (CS) students are taught the difference between the syntax and the semantics of a programming language. They learn that the syntactic constraints identify which collections of symbols are legal computer programs, while semantic constraints indicate whether a given (legal) computer program is in fact a correct (or the intended) solution to the desired problem. In order to successfully compile their programs, students must understand syntactic contracts. When CS students are introduced to methods in their beginning programming course, they learn the syntax of method declaration and invocation. In doing so, they learn about the method “signatures” (or syntactic contracts). At the same time, they learn the semantics of method invocations. Whereas the discussion of syntax involves the concrete form of the program solution, the discussion of semantics is centered around the more abstract issue of program correctness.

Although program correctness is a difficult concept for most undergraduate computer science (CS) students, we have found that correctness proofs are not beyond their capabilities. Program correctness is an inherently difficult subject, but we believe that the bigger issue is the poor attitude that students have about the usefulness of correctness proofs. We maintain that the standard methodology for introducing correctness proofs is the root of the problem, leading students to believe that program correctness is both confusing and irrelevant. We have attempted to make the concept of program correctness less confusing and more relevant by engaging our students in the task of *constructing contracts*. Introducing program correctness in this way represents a significant departure from traditional approaches to introducing program correctness.

The traditional approach to teaching program correctness (based on axiomatic semantics) is to introduce the notion of weakest preconditions using the *Assignment axiom*. After students learn the Assignment axiom, sequences (or code blocks) are covered. Our experience is that virtually all students are very comfortable using the Assignment axiom and its repeated application to sequences of statements. Confusion generally begins, however, when the semantics of conditional statements is introduced. Although the presentation usually continues to focus on backing over code to find weakest preconditions, some confusion begins with conditional statements because the presentation assumes the precondition is given. Frequently, the given precondition is not necessarily the weakest precondition, so it is necessary to introduce additional inference rules for weakening and strengthening assertions. When loops are introduced there is

more significant confusion because the focus of the discussion switches from weakest preconditions to invariants (specifically, loop invariants), and instead of working backwards through the code, students are often asked to work forward through the code. At this point, even some of the top students give up.

Our approach to teaching program correctness makes a tie between the loop invariant and the weakest precondition. A properly selected loop invariant is the weakest precondition to the loop. Furthermore, rather than asking students to “guess” the loop invariant, we encourage them to systematically “construct” the loop invariant by backing over the code. Thus, loops are processed using the same “backward partial evaluation” technique that is used for code blocks and conditional statements. Random guessing is not involved. Rather, the student is asked to infer a pattern (i.e., construct a generalized loop invariant) from a set of specific loop invariants. Knowledge of what the code is doing is not important. Students are merely identifying patterns. Thus, the process more closely models the activity involved in proving theorems using mathematical induction.

Most CS students have little difficulty understanding the need to establish behavioral contracts via preconditions and postconditions. The approach we take in the *constructing contracts* methodology is a modified version of the *design by contract* methodology [Meyer 1992a]. Rather than providing both a precondition and postcondition for each of their methods, the students are simply asked to supply a postcondition. The students then construct the appropriate precondition by working through an axiomatic proof of the correctness of the method. Thus, the proof of the method’s correctness is a side-effect of the students’ need to construct an appropriate precondition.

Most computer scientists agree that discrete mathematics is essential to both computer scientists and software engineers [Computing Curricula 2000; Bruce et al. 2003; Devlin 2003; Henderson 2003]. Over the past several years, we have made various attempts to increase the effectiveness of discrete mathematics for our students. Initially, our students took a single class in discrete mathematics taught by the mathematics faculty. Because this class was taken jointly by mathematics and CS students, computer science applications were rarely discussed. In order to introduce these applications to our students, we tried extending the discrete mathematics sequence to a full year by adding a follow-on discrete mathematics course taught by computer science faculty. The justification for this course was to make discrete mathematics more relevant by tying it to the first year programming sequence with a set of CS-complete examples [Gegg-Harrison 2001]. Although somewhat successful, we found that we needed to spend a significant amount of time reviewing many of the major topics of discrete mathematics, specifically mathematical induction. Furthermore, students continued to complain that the first discrete mathematics course was not useful.

The constructing contracts methodology is our most recent attempt to make discrete mathematics more relevant to CS students. Constructing contracts is an integral component of a new one-semester applied algorithms-based discrete mathematics course taught by computer science faculty; it replaces the year-long sequence. The course is centered around the formal specification, correctness, and complexity of computer algorithms. Logic is introduced as a formal specification language. Proof techniques are introduced as a technique for showing the validity of various properties, including correctness, of computer algorithms. Finally, counting techniques are introduced as a way of calculating the complexity of an algorithmic solution.

As a teaching tool for our applied algorithms-based discrete mathematics course, we implemented the ProVIDE system that guides our students through axiomatic proofs of correctness while they construct contracts for their Java methods [Gegg-Harrison et al.

2003]. There are three primary reasons why Java was selected as the programming language for ProVIDE:

- (1) Currently, one of the most popular programming languages for teaching introductory computer programming is Java [Java Task Force 2005]. Students at our institution are introduced to problem solving and programming in a two-semester course sequence using Java. They take the applied algorithms-based discrete mathematics course concurrently with the second semester programming course, so they already know the basics of Java.
- (2) The underlying model for Java is an imperative paradigm with state changes and looping constructs. One of the most difficult topics for our students to understand is mathematical induction. The construction of loop invariants using axiomatic semantics is a challenging, but very relevant, application of mathematical induction.
- (3) Java is an object-oriented programming language with support for exception and event handling and multitasking via threads. Although students are taught about the impact of subclassing on contracts (i.e., that a subclass can weaken the precondition or strengthen the postcondition, but a subclass cannot strengthen the precondition nor weaken the postcondition), ProVIDE currently does not support subclassing, polymorphism, exception and event handling, and multitasking; see Huizing and Kuiper [2000]; Jacobs et al. [2003]; Barnett et al. [2004] for a discussion of some of issues concerning these more advanced features of Java. Clearly, there is potential for future enhancements to ProVIDE.

In this article we present the *constructing contracts* methodology for software construction. In the next section, we define the specification language that we will use for our assertions (i.e., preconditions, postconditions, and loop invariants). The inference rules that serve as the basis for constructing contracts are given in Section 3. ProVIDE, a software construction environment that supports the constructing contracts methodology, is described in Section 4. Finally, in the last section we discuss our experience using constructing contracts and ProVIDE in the classroom.

2. SPECIFYING CONTRACTS

The first step in the construction of behavioral contracts is the definition of a specification language. Although the choice of programming language is not essential, we decided to use Java as our target programming language mainly because our student audience already had some background with Java from their first programming course. To ease the demands on our students, we selected Java **boolean** expressions as the specification language as well. Unfortunately, Java has some shortcomings as a specification language. The most obvious limitation is the lack of quantifiers. We also needed to provide mathematical collections, specifically sets and bags. In addition to these additions to the Java **boolean** expression language, we also needed to refer to the result that is returned from a non-void method invocation and to differentiate the initial and final values of variables and parameters.

In order to support return values, we added the pseudo-variable *result* to the Java **boolean** expression language. This introduces an interesting problem if the programmer wants to use the identifier *result* as an instance variable or a parameter. It is possible to differentiate the pseudo-variable *result* from an instance variable *result* by referring to the instance variable as *this.result*. Unfortunately, there is not easy way to differentiate the pseudo-variable *result* from a parameter named *result*, so we do not permit the use of *result* as the name of one of the method's parameters.

To support differentiating between initial and final values of variables and parameters, we added the pseudo-variable *old*. The pseudo-variable *old* is used analogous with the keyword *this*. For a variable or parameter named *myVariable*, we would use *old.myVariable* to refer to the original value of *myVariable* and simply *myVariable* to refer to the final value of *myVariable*. At first thought, it may seem that we need to have access to intermediate values of variables and parameters, but recall that the specification language is used for method preconditions and postconditions and class invariants. As such, there is no notion of intermediate values of variables and parameters. Preconditions are only concerned with the values of variables before the execution of the method, while postconditions are potentially concerned with defining state change in terms of original and final values of variables and parameters. Class invariants, on the other hand, are concerned with the current state, and therefore do not need the notion of original and final values.

In order to handle quantifiers and mathematical collections, we had to add additional features to the Java **boolean** expression language. We attempted to do this in a Java-like format. First of all, we needed a way to define quantified variables that captured the notion of type without requiring that they be defined before they are used. The syntax of quantified variables is *@type#num* where *type* is any legal Java type (i.e., **byte**, **short**, **int**, **long**, **float**, **double**, **boolean**, or **char**) or class (including user-defined classes). So, for example, *@int#1* would be a quantified variable of type **int**. Second, we added the notion of a bounded **boolean** expression. A bounded **boolean** expression permits a more standard representation of a bounded quantified variable. So, for example, $0 < @int\#1 \leq 10$ would bound the quantified variable *@int#1* between 1 and 10. In the notation for the quantifiers given below, it is assumed that a *<bounded_expression>* is either a bounded **boolean** expression, a standard **boolean** expression, a single unbounded quantified variable, or a set of these expressions separated by commas. Furthermore, all quantified variables must be “defined” (i.e., appear) prior to the *|* symbol.

The syntax for the universal and existential quantifiers resembles the syntax of a method invocation in Java. The formal syntax for the universal quantifier is

forall(*<bounded_expression>* | *<boolean_expression>*)

where *<bounded_expression>* is as described above and *<boolean_expression>* is an arbitrary Java **boolean** expression. As an example, the expression **forall**($0 \leq @int\#1 < \mathbf{a.length} \mid \mathbf{a}[@int\#1] > 0$) says that all of the elements in the array **a** are positive. The expression **forall**($0 \leq @int\#1 < \mathbf{a.length}, @int\#1 < @int\#2 < \mathbf{a.length} \mid \mathbf{a}[@int\#1] \leq \mathbf{a}[@int\#2]$) says that the elements in the array **a** are sorted. The expression **forall**(**elementof**(**@Card#1, hand**) | **@Card#1.suit()** == **Card.SPADES**) says that all of the cards in **hand** are spades. The formal syntax for the existential quantifier is identical to that of the universal quantifier with the “keyword” **exists** replacing **forall**:

exists(*<bounded_expression>* | *<boolean_expression>*)

The specification language has support for the mathematical collections *set* and *bag*. Mathematically, a *set* cannot contain duplicates whereas a *bag* can contain duplicates. Syntactically, however, sets and bags are identical. They have the following formal syntax in our specification language:

Set(*<simple_expression>* | *<boolean_expression>*)

Bag(*<simple_expression>* | *<boolean_expression>*)

where *<simple_expression>* is either a single unbounded quantified variable (e.g., *@int#1*) or an expression containing a quantified variable (e.g., *a[@int#1]*), and a *<boolean_expression>* is an arbitrary **boolean** expression containing the quantified variable of the *<simple_expression>*. There are several operations that are supported for these collections that have their obvious meanings:

```
elementof(<simple_expression>, <collection>)
union(<collection>, <collection>)
intersection(<collection>, <collection>)
difference(<collection>, <collection>)
```

The specification language contains a notation for implication which is nothing more than a “method call”:

```
implies(<boolean_expression>, <boolean_expression>)
```

Finally, there is also support for summation (i.e., $\sum \langle collection \rangle$) and products (i.e., $\prod \langle collection \rangle$):

```
sum(<collection>)
prod(<collection>)
```

As an example, an assertion that a method computes the factorial of its parameter **n** is **result == prod(Set(@int#1 | 0 < @int#1 && @int#1 <= n))**. The sum of the squares of the first **n** integers is expressed as **result == sum(Set(@int#1 * @int#1 | 1 <= @int#1 && @int#1 <= n))**.

Our specification language consists of standard Java **boolean** expressions extended with these expressions for the result of the method, the original value of a variable or parameter, universal and existential quantifiers, sets and bags along with their operations, summations and products. This specification language represents a minor extension to the language of Java **boolean** expressions and serves as the basis for defining and constructing behavioral contracts described in the next section.

3. CONSTRUCTING CONTRACTS

Using backward substitution to construct weakest preconditions has a long history in programming language semantics, dating back to the 1960s. Floyd proposed the notion of verifying the correctness of (flowchart) programs using the techniques of logic and *loop invariants*. He used slightly different terminology than is used today, using the term *tag* for assertions and referring to the *interpretation* of the flowchart as a set of tags added to the edges of the flowchart. He identified the *antecedent* as the proposition upon entrance to a command and *consequent* as the proposition upon exit from a command. He introduced the notion of a loop invariant as the *strongest verifiable consequent* of a cycle in the flowchart [Floyd 1967]. Hoare applied the work of Floyd to Algol-like program statements rather than flowcharts [Hoare 1969].

Dijkstra used the work of Floyd and Hoare to lay the foundation for a formal methodology of program construction, defining a predicate transformer he called the *weakest precondition* to give an axiomatic semantics to an Algol-like programming language with guarded commands and nondeterminism [Dijkstra 1976]. Dijkstra defined $wp(S, R)$ as “the weakest precondition for the initial state such that activation will certainly result in a properly terminating happening, leaving the system *S* in a final state satisfying the postcondition *R*” [Dijkstra 1976, p. 18]. Gries started where Dijkstra left off, arguing that “a program and its proof should be developed hand-in-hand, with the

proof usually leading the way” [Gries 1981, p. 164]. Backhouse also espoused infusing formal methods in program construction with his “correct-by-construction” program design [Backhouse 1986; 2003].

Based on this research into formal methods, Meyer developed a method of software engineering called *design by contract* that states that “software elements should be considered as implementations meant to satisfy well-understood specifications, not as arbitrary executable texts” [Meyer 1992a, p. 41]. Contracts are used to specify *how much* the client can expect and *how little* the server must provide. With *design by contract*, the client must guarantee certain conditions before invoking a method (i.e., the precondition) and the server then guarantees certain properties after the invocation (i.e., the postcondition). Meyer also advocates specifying the precondition and postcondition in a form that the compiler can check (e.g., in Eiffel itself) so that any violation of the contract between client and server can be detected immediately. In general, the program development process requires three stages:

- (1) specification of the precondition
- (2) specification of the postcondition
- (3) implementation of the program

Gries argues that the programmer should do the first two stages and then use formal methods along with the precondition and postcondition specifications to implement the program. In *constructing contracts*, the programmer begins by specifying the postcondition (i.e., stage 2 of the development process). After specifying the postcondition, the programmer then implements the program based on that postcondition (i.e., stage 3 of the development process). After the implementation is provided, the programmer uses formal methods along with the specified postcondition and the program implementation to construct the precondition (i.e., stage 1 of the development process). Consider the Java method for squaring an integer N using only the addition given in Figure 1.

```
/**
 * The square method computes the square of its integer
 * argument.
 *
 * @param   int N representing the number to be squared
 * @return  int representing N-squared
 *
 * @ensure  result == N*N
 */
public int square(int N) {
    if (N < 0) {
        N = -N;
    }
    int m = 0;
    int x = 0;
    int y = 1;
    while (m < N) {
        x = x + y;
        y = y + 2;
        m = m + 1;
    }
    return x;
}
```

Fig. 1. Iterative squaring method.

Although this method is relatively simple (i.e., it only involves assignment statements, a straightforward conditional statement, and a single bounded loop), it is not at all obvious that it actually computes N^2 . This makes a great example to show students, since not only is it necessary to construct its precondition, it is equally important to provide a convincing argument that it really works.

3.1 Assignment Statements

We will use axiomatic semantics to prove that a program segment is correct with respect to its precondition and postcondition. Within this framework, the semantics of an imperative programming language is defined by the assignment axiom and a set of construction rules for each of the language's control structures. Although the assignment axiom and sequence rule are constructive (i.e., they define how to construct the precondition from the given postcondition), the inference rules that are generally given for conditional statements and loops are nonconstructive (i.e., they are used to show that the given postcondition follows from the given precondition).

The goal of *constructing contracts* is to encourage CS students to construct appropriate preconditions given the postconditions they provide. As such, students are introduced to formal semantics using a constructive axiomatic semantics that is expressed as contract-enriched Hoare triples of the form $\{P\} S \{Q\}$. These triples state that P is the (constructed) weakest precondition of the block of program statements S given the postcondition Q .

The approach that we have taken using *constructing contracts* in the classroom is to introduce the assignment axiom and each inference rule with a set of simple examples and then provide the students with a complete example that contains all of the programming constructs at the very end. The assignment axiom is the fundamental axiom for imperative programming languages, infusing the notion of program state change into our logic.

$$wp(V = E, Q) = Q_{V \rightarrow E} \quad (\text{Assignment Axiom})$$

The assignment axiom defines the weakest precondition of an assignment statement as the postcondition with all the occurrences of the variable on the left hand side of the assignment statement replaced with the expression on the right hand side of the assignment statement. The assignment axiom also applies to **return** statements where the pseudo-variable *result* (which is used in postconditions to denote the value of the method) replaces V and the expression that is returned replaces E .

As an example, consider the construction of the precondition to the assignment statement $y = x + 1$; given the postcondition $(x > 0) \vee (y < 1)$. The assignment axiom defines this to be the result of replacing all occurrences of y in the postcondition with the expression $x + 1$. This produces the precondition $(x > 0) \vee ((x + 1) < 1)$ which can be simplified to $(x > 0) \vee (x < 0)$ or simply $x \neq 0$.

Note that we can extend the assignment axiom to handle arrays by expressing the substitution with a conditional expression.

$$wp(a[i] = E, Q) = Q_{a[n] \rightarrow (n \neq i) ? a[n] : E} \quad (\text{Assignment Axiom for Arrays})$$

As an example of the application of the assignment axiom on array elements, consider the construction of the precondition to the statement $a[i] = a[j] - 1$; given the postcondition

$a[j] > 0$. The assignment axiom defines this to be the result of replacing all occurrences of $a[i]$ in the postcondition with the expression $a[j]-1$. This produces the precondition $((i=j)?(a[j]-1):a[j]) > 0$ which states that if $i=j$ then $a[j] > 1$, otherwise $a[j] > 0$. For the remaining constructs, we illustrate their corresponding inference rules using the *square* method given above.

3.2 Sequences of Statements

The weakest precondition of a block of statements can be constructed by successively backing over each of the statements in the block (starting with the last statement) and using the constructed weakest precondition of each statement as the postcondition to its preceding statement. The following *sequence rule* defines the precondition construction process for a pair of statements. Repeated applications of the sequence rule can be used for blocks containing more than two statements.

$$wp(S_1; S_2, Q) = wp(S_1, wp(S_2, Q)) \quad (\text{Sequence Rule})$$

Consider the square method given in Figure 1. To construct an appropriate precondition (and prove the correctness of this method), one begins with the method postcondition:

$$result = N \times N$$

which we denote as $result = N^2$. Since **return x;** is the last statement of the method, the assignment axiom is applied (i.e., the pseudo-variable *result* is replaced with *x*) to construct its weakest precondition from the method postcondition, producing:

$$x = N^2$$

The next-to-last statement of the method is a **while** loop, so a loop invariant must be identified that is both strong enough to imply the postcondition while being weak enough to enable the student to complete the proof of the correctness of the method. Given the method postcondition and the loop guard, it is possible to construct the following loop invariant:

$$(x = N^2 - (N-m)y - (N-m)(N-m-1)) \wedge (N \geq m)$$

We introduce the *loop rule* and illustrate how to construct this loop invariant in Section 3.4. For now, let's assume that $(x = N^2 - (N-m)y - (N-m)(N-m-1)) \wedge (N \geq m)$ is an appropriate loop invariant. The *loop rule* states that a properly constructed loop invariant is an appropriate precondition to the loop. Applying the sequence rule, the loop invariant becomes the postcondition to the statement immediately preceding the **while** loop. Applying the assignment axiom to the statement **int y = 1;** produces its precondition:

$$(x = N^2 - (N-m) - (N-m)(N-m-1)) \wedge (N \geq m)$$

Backing over the **int x = 0;** statement produces:

$$(0 = N^2 - (N-m) - (N-m)(N-m-1)) \wedge (N \geq m)$$

Backing over the **int m = 0;** statement produces:

$$(0 = N^2 - (N-0) - (N-0)(N-0-1)) \wedge (N \geq 0)$$

which can be simplified to $N \geq 0$. Since the first statement of the square method is a conditional statement, we need to introduce the *conditional rule*.

3.3 Conditional Statements

The weakest precondition of a conditional statement can be constructed by considering both possible paths through the **if-else** statement. A weakest precondition can be constructed for the true case (i.e., when the guard evaluates to **true**) by considering only the statements in the then-block. The idea is to construct the weakest precondition for the statements in the then-block starting with a postcondition that is constructed by conjoining the guard with the conditional statement's postcondition. As an example, if the guard is $a \leq b$ and the conditional statement's postcondition is $a > c^2$ then we would construct the weakest precondition for the statements in the then-block starting with the postcondition $a \leq b \wedge a > c^2$. This process is repeated for the statements of the else-block starting with a postcondition that is constructed by conjoining the negation of the guard with the conditional statement's postcondition. For our example with the guard of $a \leq b$ and the conditional statement's postcondition of $a > c^2$, we would construct the weakest precondition for the statements in the else-block starting with the postcondition $a > b \wedge a > c^2$. Since there is no way to know which of these two paths will be taken on an arbitrary set of inputs, the constructed preconditions for the true case and the false case must be disjoined to form the weakest precondition for the conditional statement. The following *conditional rule* defines this process formally:

$$\begin{aligned} wp(\mathbf{if}(B) S_1 \mathbf{else} S_2, Q) \\ = (wp(S_1, Q) \wedge B) \vee (wp(S_2, Q) \wedge \neg B) \end{aligned} \quad (\text{Conditional Rule})$$

The conditional rule is used to construct the precondition $(P_1 \wedge B) \vee (P_2 \wedge \neg B)$ where P_1 is constructed from $\{P_1\} S_1 \{Q\}$ and P_2 is constructed from $\{P_2\} S_2 \{Q\}$. Note that since the weakest precondition to the “null” S_2 statement is simply the postcondition Q when the else-block is omitted then the conditional rule is used to construct the precondition $(P_1 \wedge B) \vee (Q \wedge \neg B)$ where P_1 is constructed from $\{P_1\} S_1 \{Q\}$.

Applying the conditional rule to the first statement of the square method produces:

$$((N < 0) \wedge (-N \geq 0)) \vee (\neg(N < 0) \wedge (N \geq 0))$$

which can be simplified to $(N < 0) \vee (N \geq 0)$ or simply **true**. This states that the square method is defined for all integers, which is appropriate because the purpose of the conditional is to set N to its absolute value.

3.4 Iterative Statements

Logical pretest loops are captured by the *loop invariant theorem*, which uses induction to show that a programmer-supplied *loop invariant* (i.e., a relationship between program variables that remains the same regardless of how many times the loop is executed) holds.

$$wp(\mathbf{while}(B) S, Q) = (I_k \wedge B) \vee (Q \wedge \neg B) \quad (\text{Loop Rule})$$

Given an appropriate loop invariant I_k , the *loop rule* states that it also serves as the precondition to the loop. The terminating condition τ is the condition that causes the loop to terminate, assuming it executes at all. The selection of an appropriate loop invariant is a difficult task; however, it is possible to construct an appropriate loop

invariant by starting with the loop postcondition and terminating condition and backing over the loop body until a pattern I_k can be identified.

$$I_n = \begin{cases} wp(S, Q \wedge \tau) & \text{if } n = 0 \\ wp(S, I_{n-1} \wedge B) & \text{otherwise} \end{cases} \quad (\text{Loop Invariant})$$

Given the conjunction of the loop postcondition and the terminating condition as the initial postcondition, the weakest precondition is constructed for the first backward pass through the loop body. This constructed precondition and the loop guard become the postcondition to the second backward pass through the loop body. The weakest precondition that is constructed on the second backward pass through the loop and the loop guard become the postcondition to the third backward pass through the loop, etc. After viewing the constructed preconditions for each backward pass through the loop, it is possible to identify a pattern that is the loop invariant. To ensure that the choice of a loop invariant is valid, one more backward pass through the loop should be made, beginning with the proposed loop invariant, in an attempt to show that it is possible to simplify the constructed precondition to the proposed loop invariant. This is comparable to the induction step of an induction proof.

Consider the **while** loop in the square method again. The first step in constructing a loop invariant (and therefore a precondition) for the **while** loop is to identify the loop terminating condition. For this example, the loop postcondition is $x == N*N$ and the loop terminating condition is $m == N$. Thus, one starts backing over the loop with the assertion created by their conjunction

$$(x = N^2) \wedge (m = N)$$

Backing over the statement $m = m + 1$; produces

$$(x = N^2) \wedge (m + 1 = N)$$

Since this assertion does not contain the variable y , backing over the statement $y = y + 2$; leaves the assertion unchanged:

$$(x = N^2) \wedge (m + 1 = N)$$

Backing over the statement $x = x + y$; produces

$$(x + y = N^2) \wedge (m + 1 = N)$$

In order to complete the first backward pass, one must back over the loop guard. Backing over the guard of a **while** loop is like backing over the guard of an **if-else** statement, resulting in the conjunct formed from the guard and the assertion to this point:

$$(x + y = N^2) \wedge (m + 1 = N) \wedge (m < N)$$

But this simplifies back to the previous assertion:

$$(x + y = N^2) \wedge (m + 1 = N)$$

For the second backward pass, we back over the statement block $x = x + y$; $y = y + 2$; $m = m + 1$; producing

$$(x + y + y + 2 = N^2) \wedge (m + 2 = N)$$

This can be simplified to

$$(x + 2y + 2 = N^2) \wedge (m + 2 = N)$$

We complete the second backward pass by adding the loop guard to this assertion after backing over it:

$$(x + 2y + 2 = N^2) \wedge (m + 2 = N) \wedge (m < N)$$

But this simplifies back to the previous assertion:

$$(x + 2y + 2 = N^2) \wedge (m + 2 = N)$$

Backing over the statement block **x = x + y; y = y + 2; m = m + 1**; on the third backward pass produces

$$(x + y + 2(y + 2) + 2 = N^2) \wedge (m + 3 = N)$$

This can be simplified to

$$(x + 3y + 6 = N^2) \wedge (m + 3 = N)$$

It is easy to show that a generalized loop invariant for the k^{th} backward pass is

$$(x + ky + k(k - 1) = N^2) \wedge (m + k = N)$$

By tabulating all of the weakest preconditions (i.e., specific loop invariants) for each of the backward passes through the loop body, it is usually possible to identify a generalized loop invariant. Since $m + k = N$, we know that $k = N - m$, so we can rewrite the generalized assertion as

$$(x + (N - m)y + (N - m)(N - m - 1) = N^2) \wedge (k = N - m)$$

The terms including k no longer contribute to this assertion so they can be removed, producing the loop invariant:

$$(x + (N - m)y + (N - m)(N - m - 1) = N^2) \wedge (N \geq m)$$

After this generalized loop invariant is proposed, it can be validated by taking one more backward pass through the loop, starting with this loop invariant as the postcondition and ensuring that it is possible to construct this loop invariant as the weakest precondition to this arbitrary pass through the loop (i.e., the induction case). The beauty of the *constructing contracts* methodology is that it is robust (i.e., it can be applied recursively to all programming constructs including nested **while** loops), and it is consistent with the “backward partial evaluation” approach used for the other programming constructs to construct weakest preconditions.

3.5 Function Invocations

One possibility for handling function (or *nonvoid* method) invocations is to treat them like macro expansions. When a function invocation is encountered, one can simply back over each of the statements in the function body and construct the weakest precondition starting with the postcondition to the function invocation. Although this technique will work in general, it is not very practical. Every time a function invocation is encountered, the precondition must be constructed from scratch. This is particularly undesirable for frequently used functions and functions that are invoked within loop bodies.

An alternative technique is to specify appropriate postconditions for functions independent of their invocations. We restrict our attention to side-effect free functions with postconditions of the form **result** == E (where E is an arbitrary expression). The weakest precondition of an invocation of such a function is similar to that of an assignment statement. The result of the function (i.e., E) is substituted for the function invocation and then the specified precondition of the function is conjoined to the

modified postcondition.

$$wp(f(K), Q) = Q_{f(K) \rightarrow f_{ensure}} \wedge f_{require} \quad (\text{Function Invocation Rule})$$

This rule applies to both nonrecursive and recursive function invocations. For non-recursive invocations, the precondition is constructed by applying the *function invocation rule*, and can simply be included as the constructed weakest precondition for the code segment that contains the function invocation. For recursive functions, on the other hand, we encounter the function invocation itself (and thereby need its precondition) while we are attempting to construct its precondition. Thus, recursive invocations must be handled in a similar fashion to iterative constructs. We consider the case of a nonrecursive invocation first.

3.5.1 Nonrecursive Function Invocations Consider the definition of a **sumOfSquares** method that computes the sum of the squares of the first N integers given in Figure 2. In order to construct the precondition to this method, we begin by backing over the **return sum;** statement and then we “step-in” to the **while** loop.

Backing through the loop body, we encounter the **sum = sum + square(i);** statement. On the first backward pass we have the following assertion, just prior to backing over this statement:

$$\left(\text{sum} = \sum_{k=1}^N k^2 \right) \wedge (i = N)$$

Backing over the **sum = sum + square(i);** statement produces

$$\left(\text{sum} + \text{square}(i) = \sum_{k=1}^N k^2 \right) \wedge (i = N)$$

In order to apply the function invocation rule, we need to replace the **square(i)** invocation with the square method’s postcondition (with the proper substitution of the arguments) conjoined with its precondition. Consider the definition of a *square* method (which has the method body omitted, since it is unnecessary for the discussion) in Figure 3 that has a precondition that $N > 0$ and postcondition $\text{result} = N^2$.

The function invocation rule only relies on the method’s precondition and postcondition. Substituting i for N in the postcondition for **square** produces the following assertion:

$$\left(\text{sum} + i^2 = \sum_{k=1}^N k^2 \right) \wedge (i = N) \wedge (i > 0)$$

```
/**
 * @ensure result == sum(Set(@int#1 * @int#1 |
 *                          1 <= @int#1 && @int#1 <= N))
 */
public int sumOfSquares(int N) {
    int sum = 0;
    int i = 1;
    while(i <= N) {
        sum = sum + square(i);
        ++i;
    }
    return sum;
}
```

Fig. 2. Sum of squares method.

```

/**
 * @require N > 0
 * @ensure result == N*N
 */
public int square(int N) {
    M
}

```

Fig. 3. Squaring method.

After backing over the **while** loop body several times, the following loop invariant becomes apparent:

$$\left(\text{sum} + \sum_{k=i}^N k^2 = \sum_{k=1}^N k^2 \right) \wedge (i \leq N) \wedge (i > 0)$$

After backing over the **int i = 1;** statement, we are left with $N \geq 1$ for the precondition to **sumOfSquares**.

3.5.2 Recursive Function Invocations. When we permit recursive functions, we need to handle the notion of an *invariant* as was needed for loops. The interesting characteristic about *recursion invariants* as opposed to *loop invariants* stems from the nature of these two functionally equivalent but conceptually different approaches to handling repetition. Loops by nature are constructive, they define the process of computing or constructing the desired result. Recursive definitions do not define a constructive process, but rather simply define the characteristics of the solution. As such, recursive definitions are declarative, they define the characteristics of the result rather than identifying the process of computing the result. So when we think about *invariants* for looping constructs, as compared to recursive definitions, we should expect to be modeling different aspects of the problem. For *loop invariants* we model the intermediate state of the computation, which ultimately tells us the precondition of the loop (i.e., the initial state of the computation). For recursive definitions, on the other hand, there is no notion of “state of the computation.” Rather, we are only interested in the constraints on the definition. The primary constraint on the definition of a recursive method is the precondition to the method. Thus, a *recursion invariant* is the precondition to the recursive definition.

The tricky part is that we need the method’s precondition in order to construct the recursion invariant. In other words, we need to know the recursion invariant in order to construct it. This cyclic dependence is nothing new. It mirrors the definition of a recursive method – in order to define a recursive method we need to have the method definition. In order to support the construction of the precondition to a recursive method, we need to add notation that allows us to refer to the precondition for the method on a given set of parameters. We use $\text{require}_{f(n)}$ to denote the precondition to the method f on input n . Let’s consider the recursive definition of our squaring algorithm given in Figure 4.

To construct an appropriate precondition (and prove the correctness of this method), one begins with the method postcondition:

$$\text{result} = N \times N$$

which we denote as $\text{result} = N^2$. After backing over the **return answer;** statement, we encounter the conditional statement so we apply the conditional rule, producing the

```

/**
 * @ensure result == N*N
 */
public int square(int N) {
    int answer;
    if (N == 0) {
        answer = 0;
    }
    else {
        answer = square(N - 1) + N + N - 1;
    }
    return answer;
}

```

Fig. 4. Recursive squaring method.

following equation:

$$\left((N \neq 0) \wedge \left(\text{square}(N-1) + 2N - 1 = N^2 \right) \right) \vee \left((N = 0) \wedge (0 = N^2) \right)$$

which can be simplified to

$$\left((N \neq 0) \wedge \left(\text{square}(N-1) + 2N - 1 = N^2 \right) \right) \vee (N = 0)$$

The application of the function invocation rule replaces the recursive invocation $\text{square}(N-1)$ with result of the method conjoined with the precondition to the method. Since we are constructing the precondition to the method, we replace it with $\text{require}_{\text{square}(N-1)}$ producing:

$$\left((N \neq 0) \wedge \left(((N-1)^2 + 2N - 1) = N^2 \right) \wedge \text{require}_{\text{square}(N-1)} \right) \vee (N = 0)$$

which can be simplified to

$$\left((N \neq 0) \wedge \text{require}_{\text{square}(N-1)} \right) \vee (N = 0)$$

At this point, we have the following logical recurrence relation defining the precondition of the square method:

$$\text{require}_{\text{square}(N)} = \left((N \neq 0) \wedge \text{require}_{\text{square}(N-1)} \right) \vee (N = 0)$$

We can expand this expression by recursively evaluating $\text{require}_{\text{square}(N-1)}$, producing

$$\text{require}_{\text{square}(N)} = \left((N \neq 0) \wedge \left(((N-1 \neq 0) \wedge \text{require}_{\text{square}(N-2)}) \vee (N-1 = 0) \right) \right) \vee (N = 0)$$

which simplifies to

$$\text{require}_{\text{square}(N)} = \left((N \neq 0) \wedge \left((N \neq 1) \wedge \text{require}_{\text{square}(N-2)} \vee (N = 1) \right) \right) \vee (N = 0)$$

If we expand the expression k times, this expression becomes

$$\text{require}_{\text{square}(N)} = ((N \neq 0) \wedge (N \neq 1) \wedge \dots \wedge (N \neq k-1) \wedge ((N \neq k) \wedge \text{require}_{\text{square}(N-k)})) \vee (N = k) \vee (N = k-1) \vee \dots \vee (N = 1) \vee (N = 0)$$

If we consider the case when $N = k$ then the expression $((N \neq k) \wedge \text{require}_{\text{square}(N-k)}) \vee (N = k)$ evaluates to **true**, leaving

$$\text{require}_{\text{square}(N)} = (N = k) \vee (N = k - 1) \vee \dots \vee (N = 1) \vee (N = 0)$$

which simplifies to $N \geq 0$. As this example highlights the backward partial evaluation approach espoused by the *constructing contracts* methodology it can be applied to any function whose postconditions are specified in terms of an equality defining the result that is generated.

4. PROVIDE

ProVIDE (*Program Verification Integrated Development Environment*) was developed as an aid in the classroom, giving better access to the difficult, but essential, concept of program correctness. ProVIDE is an enhanced integrated development environment (IDE) for Java that facilitates the seamless integration of analysis with the creation of computer programs. ProVIDE was originally developed by extending Netbeans, a modular standards-based open source IDE written in Java [Gegg-Harrison et al. 2003]. The current version of ProVIDE is a plug-in to the open source Eclipse IDE. It uses the Javadoc tags **@require** and **@ensure** that correspond to Eiffel's assertion constructs **require** and **ensure** [Meyer 1992b] for preconditions and postconditions, respectively.

After students have constructed and debugged their methods, ProVIDE helps them construct assertions for the **@require** tags by guiding them through proofs of correctness using axiomatic semantics to find weakest preconditions. ProVIDE starts with the

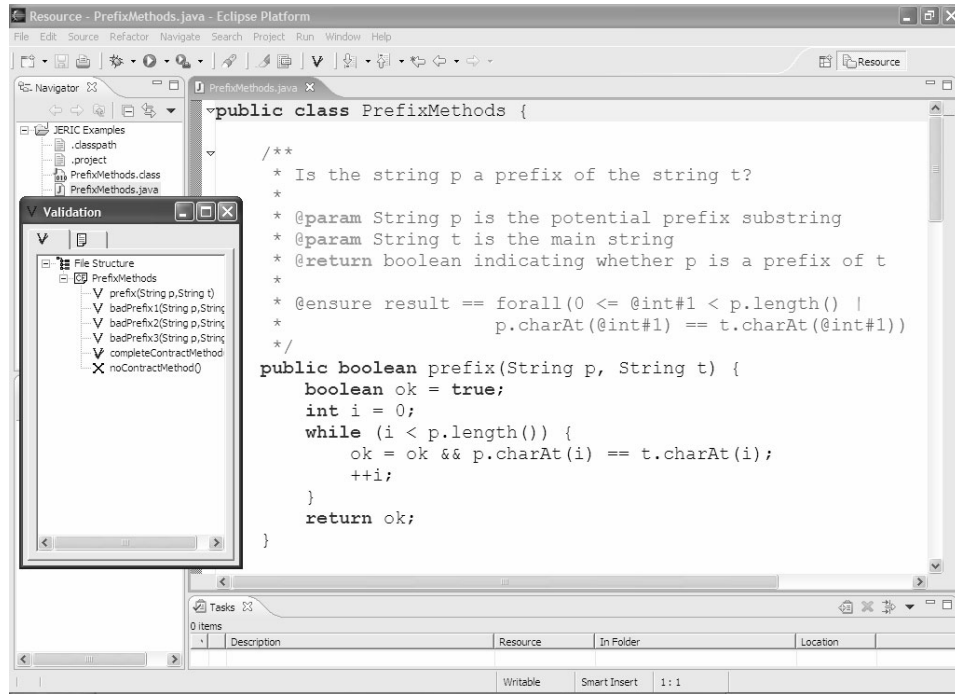



Fig. 5. ProVIDE validation initial window.

assertion, given in each method's **@ensure** tag, and helps the student construct the weakest precondition for this postcondition. This constructed precondition is added as the assertion in the method's **@require** tag. The current version of ProVIDE supports **if** statements, **if-else** statements, **while** loops, **do-while** loops, **for** loops, and method invocations using Java methods that are free of side-effects. In addition to guiding the student through the correctness proof with a set of dialog windows, ProVIDE also automatically performs the substitution for the assignment axiom. Note, however, that ProVIDE is an instructional tool. As ProVIDE does not construct preconditions for the student, but rather merely provides the framework for an appropriate correctness proof.

Once the student completes specifying the postcondition and implementing a method, he/she presses the "validate" button  and then double clicks on the method name to obtain ProVIDE's assistance with the construction of the precondition, as shown in Figure 5, where the initial dialog window of a ProVIDE session for a simple prefix method is shown.

Despite testing their programs, students still produce incorrect solutions. Many common errors can be detected and corrected while attempting to construct a precondition for a method. Beginning students often have trouble with **boolean** expressions. There are a number of ways in which a student's misunderstanding or carelessness with **boolean** expressions can lead to a faulty program. As we go through the ProVIDE session on the correct version of **prefix**, we'll also consider three potential errors: the **boolean** variable **ok** is incorrectly initialized to **false**, the **while** loop guard is incorrectly set to **i > p.length**, and the **boolean** variable **ok** is incorrectly assigned to **ok || p.charAt(i) == t.charAt(i)** in the **while** loop.

After the student double clicks on the prefix method, he/she is presented with a method validation window, as shown in Figure 6. The initial postcondition that is displayed is the postcondition that the student specified in the **@ensure** clause. ProVIDE highlights the statement that is currently under consideration. Since ProVIDE begins with

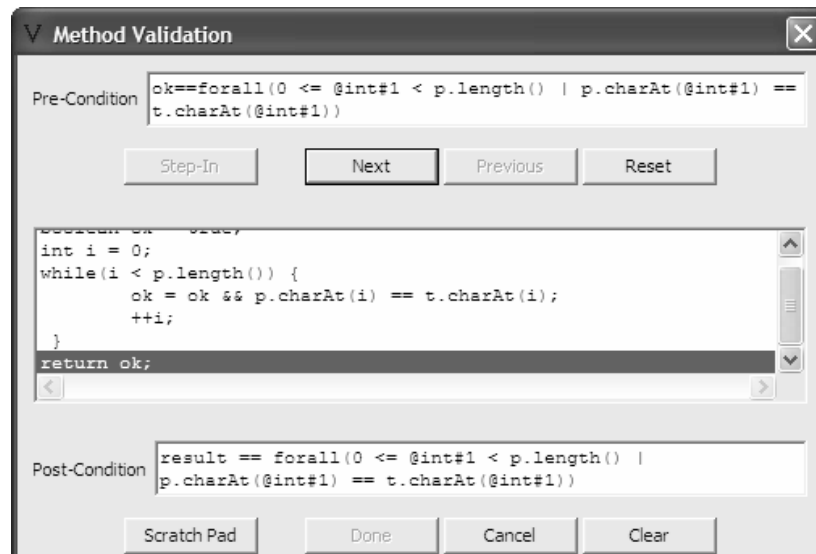


Fig. 6. Method validation window.

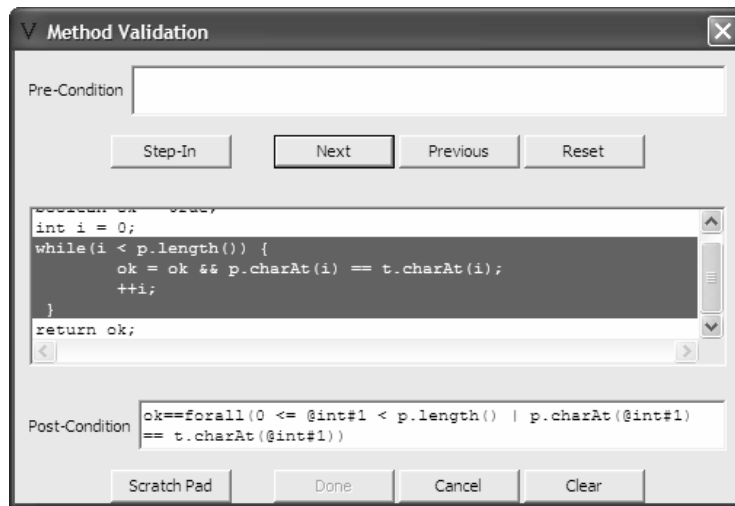


Fig. 7. While loop validation.

the last statement of the method, the current statement is the **return ok**; ProVIDE automatically fills in the pre-condition textbox to reflect the application of the assignment axiom to the postcondition with respect to the **return ok**; statement. Since there is no need to simplify the precondition, the student simply presses the **Next** button to continue to the next statement.

The next-to-last statement in the prefix method is a **while** loop which is highlighted in Figure 7. Since this is not a simple assignment statement, ProVIDE is not able to complete the precondition for the student. At this point, the student has two options. If the student already knows an appropriate precondition for the loop, then he/she can type it in the pre-condition textbox and press the **Next** button to continue to the next statement. In most cases, however, the student will not be able to produce an appropriate precondition without further analysis of the loop. ProVIDE allows the student to analyze the loop in

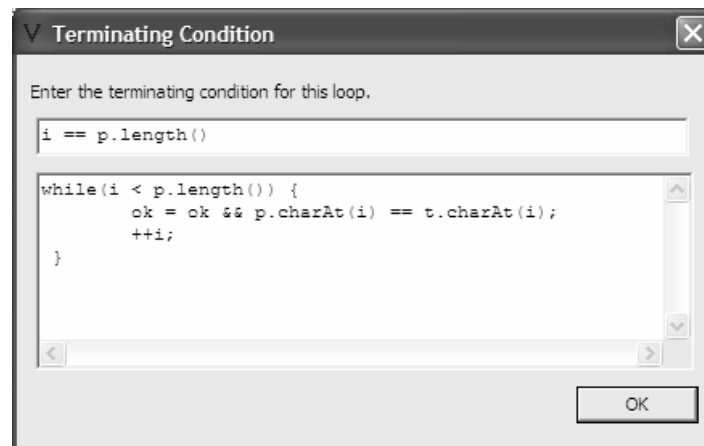


Fig. 8. Terminating condition specification.

more detail and construct an appropriate loop invariant that can be used as the precondition. The student starts this process by pressing the **Step-in** button.

In order to construct a loop invariant, the student must first enter an appropriate terminating condition. Identifying terminating conditions of loops is a very difficult problem in general. ProVIDE does not attempt to solve this problem for the students, but opens up a dialog window that contains the loop code and asks the student to fill in the terminating condition textbox shown in Figure 8. In Figure 8, the student has already filled in a terminating condition. By pressing the **OK** button, the student moves on to begin constructing the loop invariant.

At this point, the student sees the loop validation window as shown in Figure 9. At the top of this window is a text area that contains an entry for each of the loop pass invariants. Initially, it contains the assertion that must be true when the loop terminates, specifically the loop's postcondition and the terminating condition. After each backward pass through the loop, an entry is added that consists of the constructed precondition for the pass and the loop guard (which must have been true in order for the execution of the loop to take place). The second textbox (which is initially empty) is for the student's proposed (general) loop invariant. After backing through the loop a number of times, the student will use this textbox to enter his/her best "guess" at a loop invariant. ProVIDE will then force the student to validate his/her proposed loop invariant as we shall see later. In order to construct a loop invariant for the current pass through the loop, the student presses the **Next** button to sequentially back over each of the statements in the loop body.

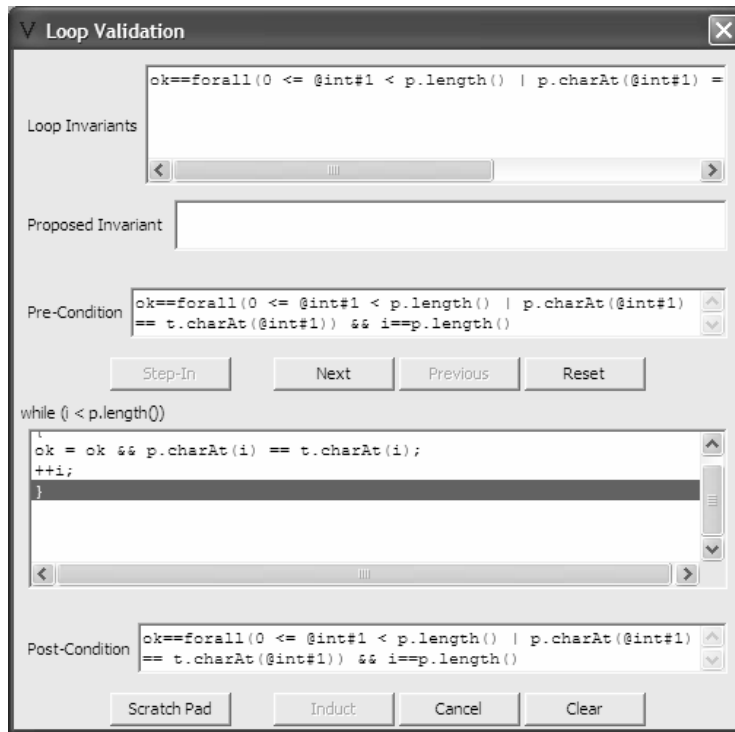


Fig. 9. Loop validation window.

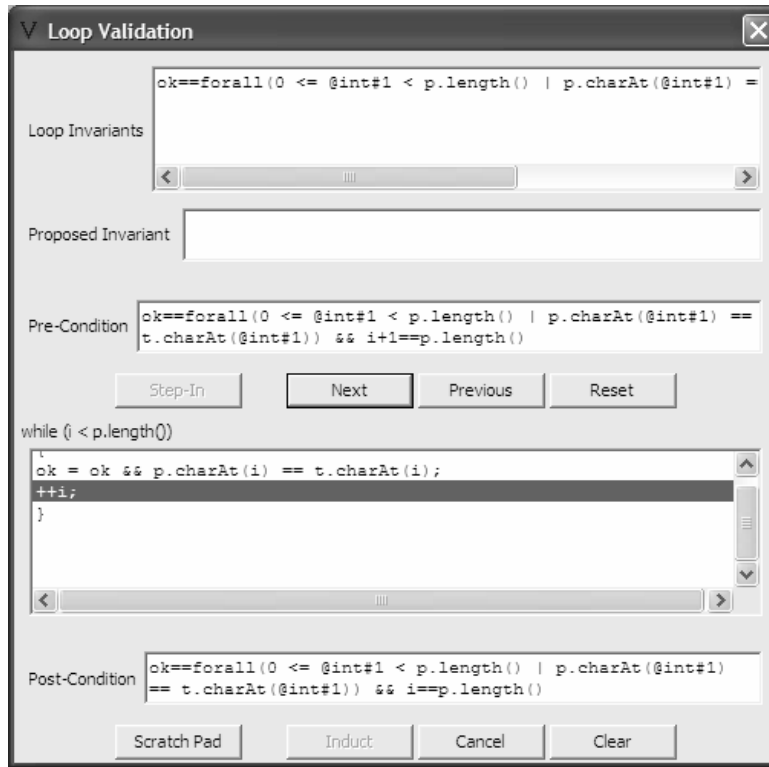


Fig. 10. Assignment axiom application.

Backing over the `++i;` statement produces a precondition that is identical to the postcondition with all occurrences of `i` replaced with `i+1`, as shown in Figure 10. At this point, the student has the option of simplifying the constructed precondition by typing in the textbox. One simplification that a student could make is to substitute the expression `i+1` for the expression `p.length()` throughout the precondition, since he/she knows that `i+1 == p.length()`. It is possible that the student makes some simplifications that he/she later regrets. If that happens, the student can always restore the precondition to its original form by pressing the **Reset** button. We'll assume that the student does not make any simplifications, however, but rather just accepts the precondition as given. To proceed, the student presses the **Next** button.

The next statement in the loop body is `ok = ok && p.charAt(i) == t.charAt(i);`. Because this statement contains method invocations, the student is given the option of leaving them as method calls or applying the *function invocation rule* as described in Section 3.5.1. In the current session, the student wants to leave the method calls and proceeds by pressing the **OK** button in the method replacement window. Since the `ok = ok && p.charAt(i) == t.charAt(i);` statement contains two method invocations, the student is presented with the method replacement window for both `p.charAt(i)` and `t.charAt(i)`. Leaving both method invocations as method calls results in replacing the variable `ok` with the expression `ok && p.charAt(i) == t.charAt(i)`, as can be seen in Figure 11.

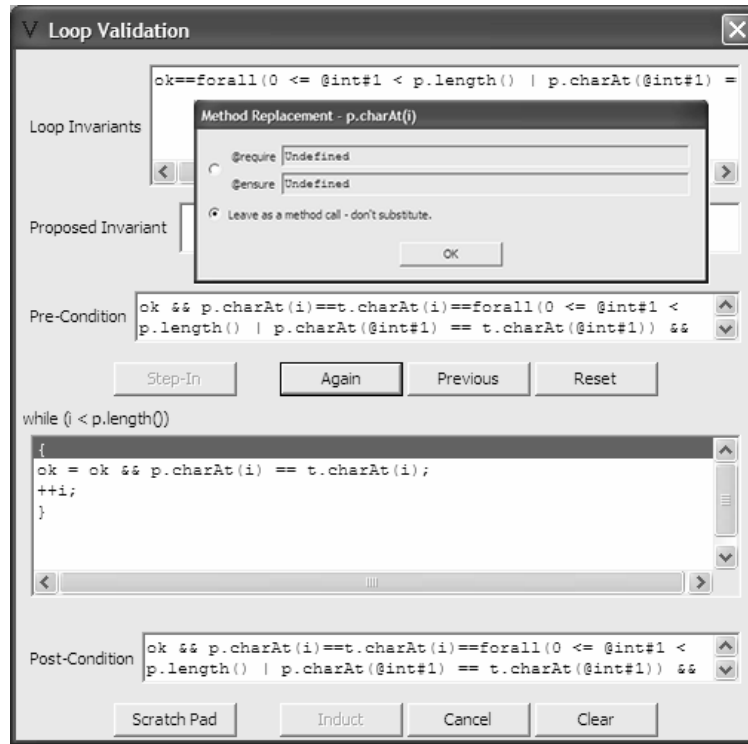


Fig. 11. Method replacement.

After completing the backward pass through the loop body, the student is given the opportunity to back through the loop again by pressing the **Again** button. When the **Again** button is pressed, the student begins another backward pass through the loop, as shown in Figure 12. The precondition that was constructed on the previous backward pass is added to the list of constructed loop invariants at the top of the window and becomes the new postcondition. The student can make as many backward passes through the loop as necessary to find a general pattern that can serve as a loop invariant.

The identification of the loop invariant can be very difficult, particularly as the assertions become very long. In order to help in that regard, ProVIDE supports a *scratch pad* that is accessible via the **Scratch Pad** button. The *scratch pad* is a separate resizable window in which the student can cut, paste, and manipulate assertions; it is shown in Figure 13. In our example, the student has copied all of the constructed loop invariants into the scratch pad. Note that the student did not simplify any of assertions so they contain expressions like `i+1+1+1` instead of `i+3`. With a little careful consideration, the student might notice that he/she can replace the expression `p.length()` in each of the assertions with an expression involving `i`. In the first assertion, `p.length()` can be replaced with `i`. One can replace `p.length()` with `i+1`, `i+2`, and `i+3`, in the second, third, and fourth assertions, respectively. In general, `p.length()` can be replaced with `i+k` on the assertion that was generated on the k^{th} backward pass through the loop. ProVIDE permits the introduction of these types of meta-variables by preceding it with the `@` symbol. Thus, the student could enter `i+@k` for the general case.

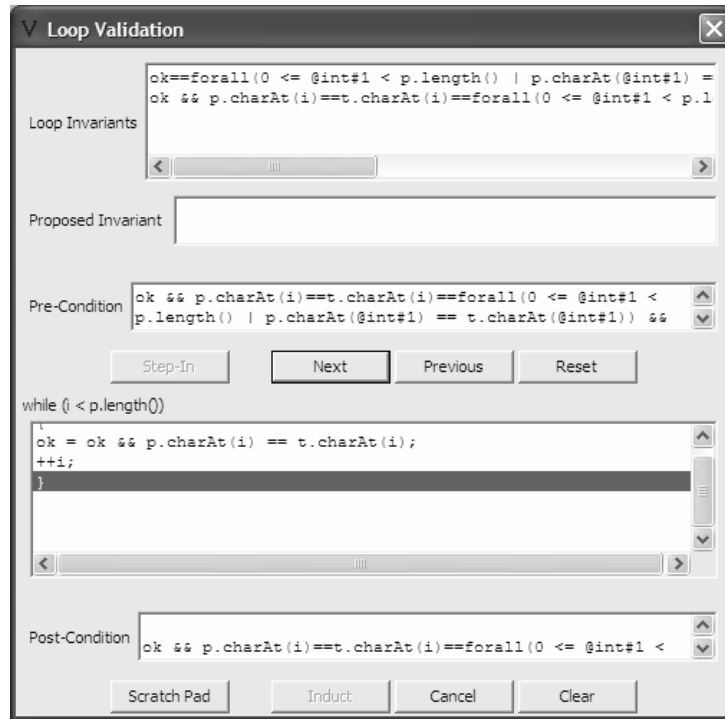


Fig. 12. Second backward pass of the loop.

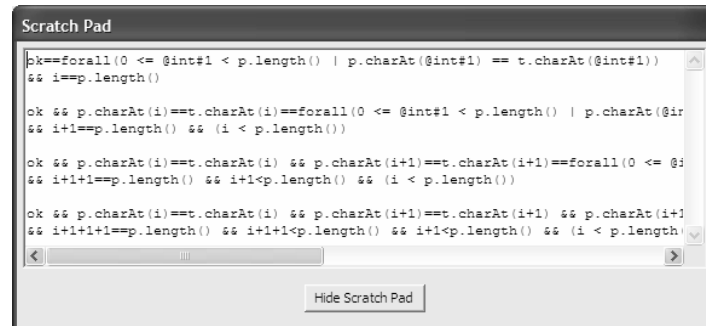


Fig. 13. Scratch pad.

It turns out that in this example, a meta-variable is not necessary because each assertion can be simplified to **ok == forall(0 <= @int#1 < i | p.charAt(@int#1) == t.charAt(@int#1)) && i <= p.length()**. This becomes the loop invariant. The student enters this assertion in the proposed invariant textbox and presses the **Induct** button as shown in Figure 14 to move to the induction step of the correctness proof.

Since the student may not come up with a correct loop invariant, ProVIDE forces the student to make one more pass through the loop, starting with the loop invariant as the postcondition and requiring that the student construct the loop invariant as a precondition. The Proposed Invariant textbox is replaced with a Must Show textbox and the student is asked to make one final backward pass through the loop. When the student

backs up to the top of the loop body, he/she must simplify the constructed precondition to be identical to the proposed loop invariant that is shown in the Must Show textbox. These steps are shown in the next two screenshots. If the student tries to simplify the constructed precondition to the proposed loop invariant and cannot, then it indicates that the proposed loop invariant is too strong. In such a case, the student would have to back up and propose another loop invariant. If, on the other hand, the student's proposed loop invariant is too weak, problems will arise later in the proof when the student uses the loop invariant as the weakest precondition to the loop. Note, however, that there is no mechanism to prevent the student from "cheating" and claiming that the constructed precondition simplifies to the loop invariant (by simply replacing the constructed precondition with the proposed loop invariant). In Figure 15, the student presses the **Done** button to continue. ProVIDE verifies that the simplified constructed precondition is identical to the proposed loop invariant. If it is not, then the student is required to either fix his/her simplification error or propose a new loop invariant. If, on the other hand, the loop invariant is acceptable, then it becomes the precondition to the **while** loop and the student continues backing over the other statements in the method. In our case, the student did indeed identify a correct loop invariant, so it becomes the precondition to the **while** loop and the student continues by pressing the **Next** button as shown in Figure 16.

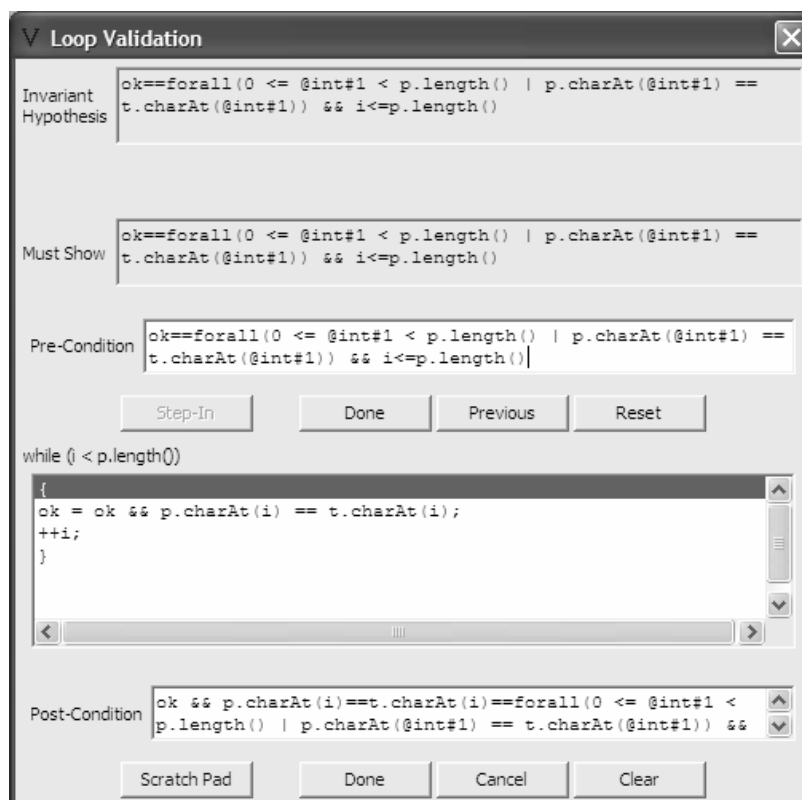


Fig. 14. Loop invariant induction

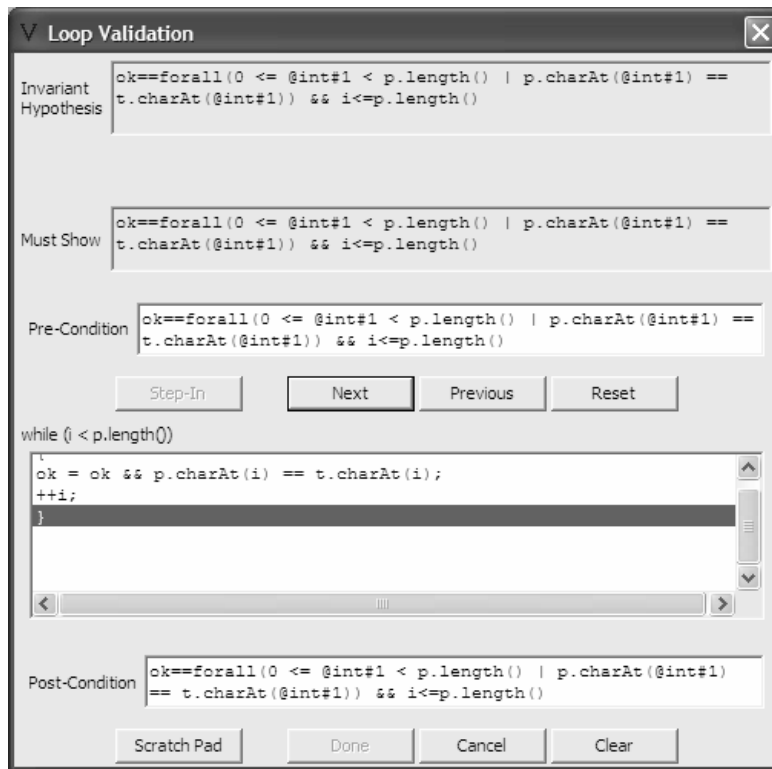


Fig. 15. Loop completion.

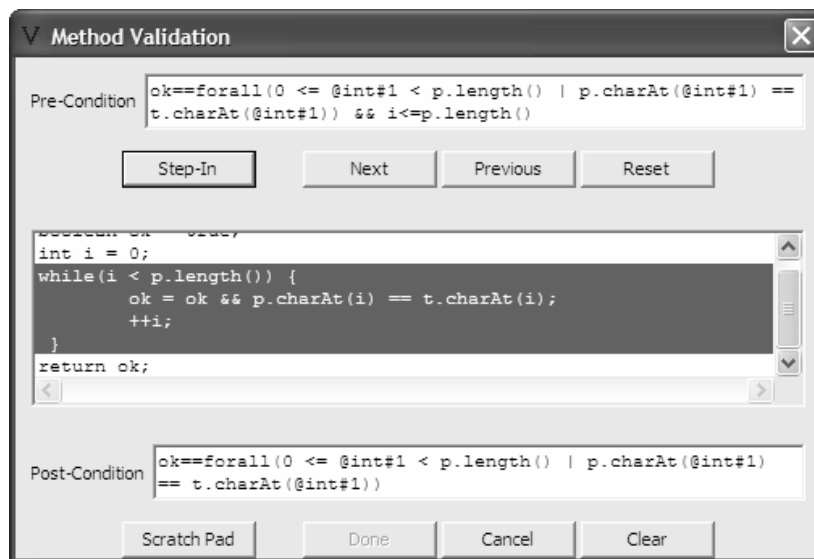


Fig. 16. Loop precondition.

Note that if the student had incorrectly entered the loop guard as $i > p.length()$, then he/she would have produced an assertion that was a contradiction while trying to construct the loop invariant. On the first backward pass through the loop body, the student would construct the following loop invariant:

```
ok == forall(0 <= @int#1 < i | p.charAt(@int#1) == t.charAt(@int#1))
&& i+1==p.length() && (i > p.length())
```

But this is a contradiction because if $i+1==p.length()$, then it is not the case that $(i > p.length())$. Upon encountering this contradiction, the student knows there is a problem somewhere. There are really only two possibilities: either the statement that incremented i (i.e., $++i$) is incorrect or the loop guard is incorrect. After careful examination of these two possibilities, the student will likely discover his/her error.

Continuing with the correct implementation, the student next backs over the simple declaration and initialization of the variable i . This expression can be simplified to $ok == true$, since the bounds on the **forall** expression restrict $@int\#1$ to no values and expression $0 <= p.length()$ is always true. The student would type in $ok == true$ as the precondition and press the **Next** button to continue. At any time during the validation process, the student always has **Previous**, **Clear**, and **Cancel** buttons. The **Previous** button moves back to the previous step in the validation process. All information is saved, so the student will see his/her previous work the next time he/she attempts to validate the same method. The **Clear** button is used to wipe out the saved steps in the validation process for the current method. The **Cancel** button cancels the current validation and returns the student to the initial Validation dialog window. Backing over the **boolean ok = true;** true statement produces a precondition of $true == true$, which can simply be simplified to **true**.

Note that at this point, if the student had incorrectly initialized the variable ok to **false**, and so another contradiction would arise. Since the **boolean ok = true;** statement is

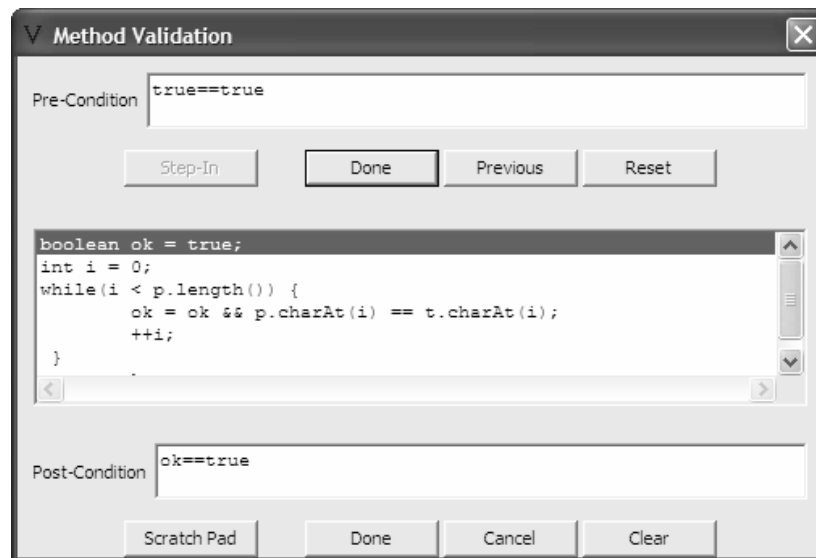


Fig. 17. Method validation completion.

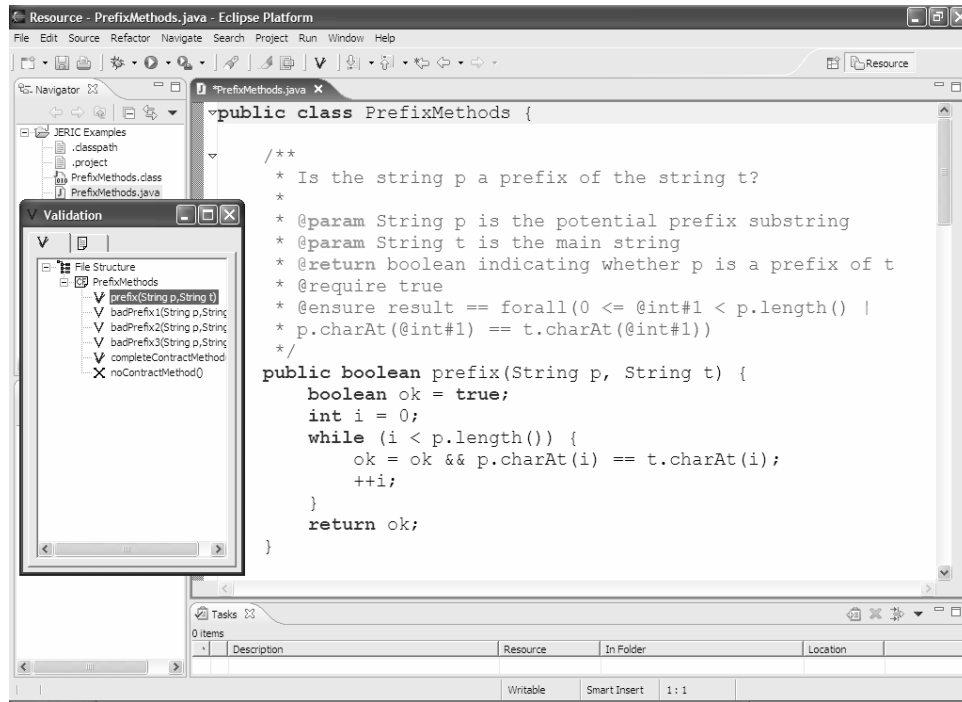


Fig. 18. ProVIDE validation final window

the first statement of the method, the **Next** button is replaced by a **Done** button in Figure 17. After the student simplifies the precondition to **true** and presses the **Done** button, ProVIDE adds the newly constructed precondition to the method with the **@requires** Javadoc tag, as shown in Figure 18.

Interestingly, if the student's mistake was to code the `ok = ok && p.charAt(i) == t.charAt(i);` statement as `ok = ok || p.charAt(i) == t.charAt(i);` then the method would actually validate, but would result in the following precondition:

forall(0 <= @int#1 < p.length() | p.charAt(@int#1) == t.charAt(@int#1))

This precondition says that the **prefix** method is correct as long as **p** is a prefix of **t**. This precondition is constructed because the method always returns **true**. Clearly, this precondition is too restrictive. Determining if **p** is a prefix of **t** is the responsibility of the **prefix** method, not the client (or invoker of **prefix**). ProVIDE not only helps its students construct preconditions for their methods, it also helps them identify incorrect programs when they construct contradictions and unreasonable preconditions.

5. CONCLUSION

Mathematics is an integral part of all science, engineering, and technological education. In particular, mathematics is fundamental to computer science. In order to better integrate some of the essential mathematics concepts into the introductory CS curriculum, we created the *constructing contracts* methodology and implemented ProVIDE, an enhanced integrated development environment (IDE) that enables students to analyze their computer programs (in terms of their correctness) while they are creating them. We are optimistic that this is having a positive impact on our students, as we have seen a

heightened level of enthusiasm and interest in the concept of program correctness as a result of *constructing contracts* and the ProVIDE system.

Preliminary results indicate that ProVIDE enhances student learning by providing the seamless integration of analysis with the creation of computer programs. Our optimism stems from our experience in introducing the concept of program correctness in Spring 2004 to students in the second semester programming classes using *constructing contracts*. All the students were introduced to program correctness with the same set of notes that were developed as part of the model curriculum. One group of students received a lecture presentation of the material and was then asked to complete an associated problem set. A second group of students received the same lecture presentation of the material, saw a software demonstration of ProVIDE on a simple function that contained a conditional statement and a **while** loop, and was then asked to complete the same problem set without the assistance of ProVIDE. The final group of students received the same lecture presentation of the material, saw a software demonstration of ProVIDE on a simple function that contained a conditional statement and a **while** loop, and was then asked to complete the same problem set using ProVIDE. Unfortunately, we found that introductory CS students still found the concept of program correctness difficult. We tried using *constructing contracts* and ProVIDE in our upper-level programming languages class (with a set of students who had not previously been exposed to the concept); this was much more successful than with the second-semester programming class.

Although students still struggle with the concept of program correctness, the results of a survey given to the students indicate that *constructing contracts* and the ProVIDE system are having a positive impact on their attitudes. Nearly all of the students (93%) appreciated the role of program correctness, indicating that they believe that program verification is better than program testing and that program verification is effective as a debugging technique. Eighty-two percent of the students surveyed believed that our technique for program correctness was indeed general enough to be applied to any computer program. These results are significant because they indicate that students are “buying in” to the notion that program correctness is an important concept. Being able to convince the students that what they are learning is worthwhile is a major step in the learning process. Although the majority of the students still felt that the material was at least “somewhat difficult” to understand and apply, 35% of the students who saw the software demonstration of ProVIDE felt the material was at least “fairly easy” to understand and apply, as opposed to only 10% of the students who did not see the software demonstration of ProVIDE. Thus, it appears that ProVIDE increases the students’ “comfort level” of the material.

ACKNOWLEDGMENT

The author would like to acknowledge the contributions of Gary Bunce, Rebecca Ganetzky, Christina Olson, Jeremy Scherer, Keith Swanson, and Josh Wilson. The final version of this paper is significantly improved as a result of some very constructive comments from the anonymous reviewers.

REFERENCES

- ACM JAVA Task Force. 2005. Java Task Force Report: First Public Draft. Internet. Feb.1, 2005: <http://cs.stanford.edu/~eroberts/jtf/>.
- ACM/IEEE Joint Task Force On Computing Curricula. 2001. Computing Curricula 2001: Computer Science. Internet. Dec. 15, 2001. <http://www.sigcse.org/cc2001/>.
- BACKHOUSE, R.W. 1986. *Program Construction and Verification*. Prentice Hall, Englewood Cliffs, N.J.

- BACKHOUSE, R.W. 2003. *Program Construction: Calculating Implementations from Specifications*. John Wiley, New York.
- BARNETT, M., DELINE, R. FAHNDRICH, M., LEINO, K.R.M., AND SCHULTE, W. 2004. Verification of object-oriented program with invariants. *J. Object Technol.* 3, 6, 27-56.
- BEUGNARD, A., JEZEQUEL, J.M., PLOUZEAU, N., AND WATKINS, D. 1999. Making components contract aware. *IEEE Computer* 32, 7, 38-45.
- BRUCE, K.B., DRYSDALE, S.R.L., KELEMAN, C., AND TUCKER, A. 2003. Why math? *Commun. ACM* 49, 9, 40-44.
- DEVLIN, K. 2003. Why universities require computer science students to take math. *Commun. ACM* 49, 9, 37-39.
- DIJKSTRA, E.W. 1976. *A Discipline of Programming*. Prentice Hall, Upper Saddle River, N.J.
- FLOYD, R.W. 1967. Assigning meaning to programs. In *Proceedings of the 19th Symposium on Applied Mathematics (Mathematical Aspects of Computer Science)*, American Mathematical Society, Providence, RI.
- GEGG-HARRISON, T.S. 2001. Egyptian numbers: A CS-complete example. In *Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education (SIGCSE 2001)*, ACM, New York, 268-272.
- GEGG-HARRISON, T.S., BUNCE, G.R., GANETZKY, R.D., OLSON, C.M., AND WILSON, J.D. 2003. Studying program correctness by constructing contracts. In *Proceedings of the 8th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE 2003)*, ACM, New York, 129-133.
- HUIZING, K. AND KUIPER, R. 2000. Verification of object-oriented programs using class invariants. In *Proceedings of the 3rd International Conference on Fundamental Approaches of Software Engineering (FASE 2000)*, Springer-Verlag, New York, 208-221.
- GRIES, D. 1981. *The Science of Programming*. Springer-Verlag, New York.
- HENDERSON, P.B. 2003. Mathematical reasoning in software engineering education. *Commun. ACM* 49, 9, 45-50.
- HOARE, C.A.R. 1969. An axiomatic basis for computer programming. *Commun. ACM* 12, 10, 576-580.
- JACOBS, B., KINIRY, J., AND WARNIER, M. 2003. Java program verification challenges. In *Proceedings of the 1st International Symposium on Formal Methods for Components and Objects (FMCO 2002)*, Springer-Verlag, New York, 202-219.
- MEYER, B. 1992a. Applying "design by contract." *IEEE Computer* 25, 10, 40-51.
- MEYER, B. 1992b. *Eiffel: The Language*. Prentice Hall, Englewood Cliffs, N.J.

Received May 2004; revised August 2005; accepted January 2006