# Invariant Assertions for Program Proofs

Mark Ardis and Curtis Clifton

April 2010*

## 1 Introduction

There are a few different methods for formally proving that a program agrees with its specification. The first method we will examine is the *invariant assertion* method. The method was first proposed for flowcharts by Robert Floyd [3] and then adapted for program code by Tony Hoare [5]. Edsger Dijkstra developed another variant of the method a few years later [1, 2].

Invariant assertions are statements that are always true (hence the name invariant), no matter what values are taken by the variables in the program. Thus, the text of the program can be annotated with these assertions, for they will be true whenever control passes through them. Some programming languages, like C and recent versions of Java [4], even have special constructs for including assertions and checking their validity during execution. This document uses the syntax of the Java Modeling Language (JML) for stating assertions [6]. JML assertions are written using a special style of comments, beginning with `//@` or wrapped in `/*@ ... @*/`.

Here is an example of an assignment statement with assertions before and after:

```
    //@ assert n == 0;
i = 0;
    //@ assert n == i;
```

The first assertion, `n == 0`, is called the *pre-condition* of the statement, and the second assertion is called the *post-condition*. We can also refer to pre-conditions and post-conditions of blocks of statements.

The invariant assertion method of proof consists of three steps:

1. Write the specification of a program in terms of a pre-condition and post-condition for the program.

*Based on an original document by Mark Ardis, September 2005

2. Show that the program preserves those assertions by applying rules of inference to further annotate the program.

3. Show that any loops or recursive function calls always terminate.

If only the first two steps are performed the program is said to have been proven *partially correct*. It is not deemed *totally correct* unless all three steps are performed. A partially correct program is proven to satisfy its specification if it terminates. A totally correct program always terminates.

As we will see, some of the steps of this method are fairly mechanical (plug-and-chug), while others require some insight into the meaning of the program. Still other steps do not require any special knowledge of the program, but they do require some facility with logical inference. There are tools that can partially automate this proof process, but (at least initially) we will focus on the techniques not the tools.

This paper describes a basic set of rules that we'll use for proving program properties. It then gives an example of applying these rules.

## 2 Rules of Inference

For each programming language construct there is one rule that explains its meaning, or semantics. There are also rules for composing statements and assertions. Initially we will use only assignment statements and four control structures: sequencing, if-then, if-then-else, and while. Later we will look at procedure calls.

### 2.1 Assignment

Here is the first rule:

| **Statement** | **Rule** | **Informal Meaning** |
|---|---|---|
| Assignment | //@ assert $P(e)$;<br>$v = e$;<br>//@ assert $P(v)$; | Whatever is true about the expression $e$ before the assignment statement (as given by the predicate $P$) is true about the variable $v$ afterward. |

The assignment rule allows us to reason about assignment statements. The pre-condition is a predicate $P$ about the expression $e$ that appears on the right-hand side of the statement. The post-condition is the same predicate $P$ about the variable $v$ that appears on the left-hand side of the statement. In the example we saw earlier:

```
    //@ assert n == 0;
i = 0;
    //@ assert n == i;
```

the predicate $P(x)$ is `n == x`. (Think of this as "$P$(_) is `n == _`", where $x$ represents the blank to be filled in.) The first instance of the predicate in the example is $P(0)$, and the second instance is $P$(i). One way of reading this example is to say that if `n == 0` holds when we reach the assignment statement, then `n == i` will be true after execution of the statement.

## 2.2 Sequencing

The next rule handles sequences of assertions.

| **Statement** | **Rule** | **Informal Meaning** |
|---|---|---|
| Weakening | If $P$ `==>` $Q$ then<br>`//@ assert` $P$;<br>`//@ assert` $Q$;<br>is valid. | Consecutive assertions in a proof of correctness may be "weakened", but not strengthened.<br><br>We say that $Q$ is *weaker* than $P$, because it says less. That is, $P$ is strong enough that knowing it is true is sufficient to know that $Q$ is true, $P$ `==>` $Q$. On the other hand, knowing that $Q$ was true, we couldn't necessarily conclude that $P$ was true. |

The weakening rule is commonly used at the beginning and end of proofs. At the beginning, we can use the rule to connect a given pre-condition to the first statement in the program. At the beginning of a proof, we can use the rule to connect the last statement in the program to a given post-condition.

The other use of the weakening rule is for joining the proofs of two consecutive statements, as codified in the rule below.

| Statement | Rule | Informal Meaning |
|---|---|---|
| Composition | If:<br>`//@ assert` $P_1$;<br>$S_1$;<br>`//@ assert` $Q_1$;<br>and $Q_1$ ==> $P_2$ and<br>`//@ assert` $P_2$;<br>$S_2$;<br>`//@ assert` $Q_2$;<br>then<br>`//@ assert` $P_1$;<br>$S_1$;<br>$S_2$;<br>`//@ assert` $Q_2$; | If the post-condition of $S_1$ implies the pre-condition of $S_2$, then the proofs can be "glued" together. |

This rule can be thought of as a way to compose proofs. It formalizes the observation that an assertion that appears above another must imply it logically.

This same idea lets us relax conditions moving down a program (or equivalently, strengthen the condition moving up). For example, suppose at some point in a program we have:

```
//@ assert n == 0 && m > 0;
```

The condition `n == 0 && m > 0` implies `n == 0`, so we can write:

```
    //@ assert n == 0 && m > 0;
    //@ assert n == 0;
i = 0;
    //@ assert n == i;
```

## 2.3 If-then-else

Control statements with branching are a bit more complicated, since the rule needs to hold regardless of which path through the program is actually taken at run time.

| Statement | Rule | Informal Meaning |
|---|---|---|
| If-then-else | If:<br>`//@ assert` $P_1$ `&&` $B$;<br>$S_1$;<br>`//@ assert` $Q$;<br>where $P$ `&&` $B$ `==>` $P_1$<br>and<br>`//@ assert` $P_2$ `&&` `!`$B$;<br>$S_2$;<br>`//@ assert` $Q$;<br>where $P$ `&&` `!`$B$ `==>` $P_2$<br>then:<br>`//@ assert` $P$;<br>`if (` $B$ `) {`<br>$S_1$;<br>`} else {`<br>$S_2$;<br>`}`<br>`//@ assert` $Q$; | Suppose we can show that the body of the then-clause, $S_1$, satisfies its pre- and post-conditions ($P_1$ and $Q$ resp.) by assuming the if-statement's pre-condition, $P$, and conditional expression $B$ hold. Also suppose we can show that the body of the else-clause, $S_2$, satisfies its pre- and post-conditions ($P_2$ and $Q$ resp.) by assuming the if-statement's pre-condition, $P$, holds but $B$ *does not*. Then the if-statement satisfies the given pre- and post-conditions.<br><br>In general, the pre-condition, $P$, of the if-statement is $(B$ `==>` $P_1)$ `&&` $(!B$ `==>` $P_2)$, which ensures that $P$ `&&` $B$ `==>` $P_1$ and $P$ `&&` `!`$B$ `==>` $P_2$. Stronger pre-conditions may be used, but this general form is used when finding the "weakest pre-condition", i.e., when working backward from $Q$ to derive $P$. |

This rule subdivides the problem of reasoning about an if-then-else statement into two new problems: (i) show that the body of the then-clause satisfies some pre- and post-condition, and (ii) show that the body of the else-clause satisfies a related set of pre- and post-conditions. If both proofs hold, the rule draws a conclusion about the entire if-statement.

Suppose we had proven (using our assignment and sequencing rules from earlier):

```
    //@ assert (x + y <= 2 * x) && (2 * x < 10) && (x > y);
    //@ assert (x + y <= 2 * x) && (2 * x < 10);
z = 2 * x;
    //@ assert (x + y <= z) && (z < 10);
```

and we had proven:

```
    //@ assert (x + y <= 2 * y) && (2 * y < 10) && (x <= y);
    //@ assert (x + y <= 2 * y) && (2 * y < 10);
z = 2 * y;
    //@ assert (x + y <= z) && (z < 10);
```

Then we would be able to conclude:

```
    /*@
      @ assert (x > y)  ==> (x + y <= 2 * x) && (2 * x < 10) &&
      @        (x <= y) ==> (x + y <= 2 * y) && (2 * y < 10);
      @*/
if (x > y) {
    z = 2 * x;
} else {
    z = 2 * y;
}
    //@ assert (x + y <= z) && (z < 10);
```

Here is how the pieces match up:

| | | |
|---|---|---|
| $P_1$ | is | (x + y <= 2 * x) && (2 * x < 10) |
| $P_2$ | is | (x + y <= 2 * y) && (2 * y < 10) |
| $Q$ | is | (x + y <= z) && (z < 10) |
| $B$ | is | (x > y) |
| $!B$ | is | (x <= y) |

### 2.4 If-then

The rule for if-then statements is similar, except that there is no else clause.

| Statement | Rule | Informal Meaning |
|---|---|---|
| If-then | If:<br>`//@ assert` $P_1$;<br>$S$;<br>`//@ assert` $Q$;<br>where $P$ && $B$ ==> $P_1$<br>and $P$ && !$B$ ==> $Q$<br>then:<br>`//@ assert` $P$;<br>if ( $B$ ) {<br>$S$;<br>}<br>`//@ assert` $Q$; | Suppose we can show that the body of the then-clause, $S$, satisfies its pre- and post-conditions ($P_1$ and $Q$ resp.) by assuming the if-statement's pre-condition, $P$, and conditional expression $B$ hold. Also suppose we can show that post-condition, $Q$, holds by assuming that $P$ holds but that $B$ *does not*. Then the if-statement satisfies the given pre- and post-conditions.<br><br>As with the if-then-else rule, there is a general form for $P$ that can be used when working backwards to find the weakest pre-condition. In general, $P$ is $(B \texttt{ ==> } P_1)$ && $(!B \texttt{ ==> } Q)$. |

There are still two sub-problems to solve here: one for the then-clause and one for skipping it. If both can be solved, then the rule draws a conclusion about the entire if-statement.

### 2.5 While

The while rule is similar to the if-then rule, but it has to handle the cases when the loop body is executed more then once.

| Statement | Rule | Informal Meaning |
|---|---|---|
| While | If:<br>`//@ assert` $P$ && $B$;<br>$S$;<br>`//@ assert` $P$;<br>and $P$ && !$B$ ==> $Q$<br>then:<br>`//@ assert` $P$;<br>while ( $B$ ) {<br>$S$;<br>}<br>`//@ assert` $Q$; | Suppose we can show that the body of the loop, $S$, satisfies its pre- and post-conditions. Also suppose we can show that post-condition, $Q$, holds by assuming that $P$ holds but that $B$ *does not*. Then the while-statement satisfies the given pre- and post-conditions.<br><br>The pre-condition for the body includes $B$, since it is only executed when $B$ holds. The pre-condition for the while-statement does not include $B$, since the loop may not be executed in some cases. For some iterations, the loop may maintain $B$, but it must eventually yield !$B$ (or else the loop doesn't end). |

The while rule uses a predicate $P$ that remains true for any number of iterations of the loop. It is true before the loop, after each iteration of the loop (if any), and after the loop is complete. For that reason $P$ is called the *loop invariant*. Because we also know that the loop condition $B$ will be false when the loop finishes, we can strengthen the post-condition of the while statement to include it.

The example in Section 6 shows how we can use the while rule in a proof.

## 3 Termination

In order to prove total correctness of a program with its specification, you must show that the program always terminates. There are two programming language elements that could cause trouble: loops and recursive functions. We will defer recursive functions until we build up some notation and rules for functions.

How can you show that a loop always terminates? You must identify some expression, called a *bound function*, whose value:

- changes after each iteration of the loop
- always changes in the same direction
- is bounded by the loop condition

If the expression gets bigger each time through the loop then we say that it is *strictly increasing*, and if it gets smaller each time then we say that it is *strictly decreasing*.

In order to find a bound function you need to first examine the loop condition to see what it can bound. Whatever variables appear in the condition are potential components of the bound function. Next you need to examine the body of the loop to see which variables change every time through the loop. Finally, you need to compose an expression consisting of variables that change and appear in the loop condition. You may need to combine several variables in order to get something that is strictly increasing or strictly decreasing.

## 4 Additional Notation

It helps to have some additional notation for describing arrays and other data structures. JML provides some notation for this including that shown in the table below.

| Expression | Meaning |
|---|---|
| (\forall int i; lo <= i && i <= hi; $P$(a[i]) ) | This expression is true if the predicate $P$ holds for every element in the array a in the range a[lo], ..., a[hi]. If the range is empty (i.e. hi < lo), then the expression is vacuously true. |
| (\exists int i; lo <= i && i <= hi; $P$(a[i]) ) | This expression is true if the predicate $P$ holds for some element in the array a in the range a[lo], ..., a[hi]. If the range is empty (i.e. hi < lo), then the expression is vacuously false. |
| (\sum int i; lo <= i && i <= hi; a[i] ) | This expression evaluates to the sum of every element of the array a from a[lo] through a[hi]. If the range is empty (i.e. hi < lo), then the expression evaluates to 0. |
| (\min int i; lo <= i && i <= hi; $F$(i) ) | This expression evaluates to the minimum of the function $F$ over (the integers in) the range from lo to hi. If the range is empty (i.e. hi < lo), then the expression evaluates to negative infinity or Integer.MIN_VALUE, depending on whether $F$ is floating point or integer according to Java's typing rules. |
| (\max int i; lo <= i && i <= hi; $F$(i) ) | This expression evaluates to the maximum of the function $F$ over (the integers in) the range from lo to hi. If the range is empty (i.e. hi < lo), then the expression evaluates to positive infinity or Integer.MAX_VALUE, depending on whether $F$ is floating point or integer according to Java's typing rules. |

The use of the backslash in the keywords \forall and \sum is a peculiar feature of JML. Can you see why it might be necessary? What if we were trying to prove a property about a program where the developer had used sum as a variable name?

Here is an example of an assertion that says that an array, scores, is sorted:

```
//@ assert (\forall int i; 0 <= i && i < arr.length - 1; scores[i] <= scores[i+1]);
```

This example puts a bound on the average of the square roots of the elements of the array named sig.

```
//@ assert (\sum int i; 0 <= i && i < sig.length; Math.sqrt(sig[i]) ) / sig.length < 1.0;
```

## 5 Constructing Proofs of Programs

Invariant assertion proofs of programs are developed bottom-up (or backwards from the end). Starting with the post-condition for the last statement in the program, you apply rules of inference to calculate the pre-condition of each statement. Eventually you should reach the pre-condition for the very first statement. If your calculated pre-condition is implied by the pre-condition for the program, then you have constructed the partial proof of the program's correctness.

Then, for each loop, you construct a bound function and demonstrate that it obeys the appropriate properties. That is, either you show that it is strictly increasing and bounded above by the loop condition, or you show that it is strictly decreasing and bounded below.

## 6 Example

Here is a simple program that calculates the product of two integers, one of which must be greater than zero. Notice how the program's pre- and post-condition formally encode this informal specification.

```
    //@ assert b > 0;
r = 0;
c = b;
while (c > 0) {
    r = r + a;
    c = c - 1;
}
    //@ assert r == a * b;
```

Remember that we're trying to work backward from the post-condition to arrive at the pre-condition. So, to prove this program correct the first step is to rewrite the post-condition for the program into one appropriate for the while loop. The inference rule for while loops requires that the post-condition, $Q$, be such that $P$ && $!B$ ==> $Q$. The simplest approach for $Q$ is to just let it be $P$ && $!B$, where $P$ is the loop invariant and $B$ is the loop condition. In this case $B$ is c > 0, so $!B$ is c <= 0.

Determining the loop invariant is typically the most challenging part of a correctness proof. It typically involves a bit of informed guess work. One piece of leverage that we have is to consider what happens to the variables in the loop condition as the loop runs.

From informal study of the code we can see that c == 0 when the loop terminates. (It's greater than zero at the start of the loop and decreases by one with each iteration. The loop terminates

when c is non-positive.) We want to have $!B$ appear in the loop post-condition, So, we write c == 0 as two inequalities, c >= 0 && c <= 0. These two inequalities say nothing more or less than c == 0, but the second inequality is $!B$. We can "back up" from the program's post-condition to a new assertion involving these inequalities (shown in bold):

```
    //@ assert b > 0;
r = 0;
c = b;
while (c > 0) {
    r = r + a;
    c = c - 1;
}
    //@ assert r == a * b && c >= 0 && c <= 0;
    //@ assert r == a * b;
```

There are two key things to notice here: (i) We haven't yet *proven* that this new assertion holds after the loop. We're just *guessing* that it will based on our knowledge of the code. (ii) The new assertion *implies* the program's post-condition, so we're following the sequencing rule given in Section 2.2.

Next we continue trying to guess the loop invariant $P$. Consider what variables are changed in the loop body. Besides c, we see that r is incremented by a each time through the loop. How many times has r been incremented at any point in time? The number of increments is just the difference between b and c. And this is true before the loop is even executed once. (Before the loop is executed the difference between b and c is zero and we've added a to r zero times.) So, we rewrite the post-condition to use that fact.

```
1     //@ assert b > 0;
2  r = 0;
3  c = b;
4  while (c > 0) {
5     r = r + a;
6     c = c - 1;
7  }
8     //@ assert r == a * (b - c) && c >= 0 && c <= 0;
9     //@ assert r == a * b && c >= 0 && c <= 0;
10    //@ assert r == a * b;
```

Have we broken any rules here? Does the assertion on line 9 follow from the assertion on line 8? It does! Recall that the last two inequalities tell us that c == 0. It's clearer to rewrite the assertion in line 9 to make this more explicit:

```
1       //@ assert b > 0;
2   r = 0;
3   c = b;
4   while (c > 0) {
5       r = r + a;
6       c = c - 1;
7   }
8       //@ assert r == a * (b - c) && c >= 0 && c <= 0;
9       //@ assert r == a * (b - c) && c == 0;
10      //@ assert r == a * b;
```

Line 8 gives a loop post-condition in the form $P$ && $!B$, where the loop invariant $P$ is:

$$\texttt{r == a * (b - c) \&\& c >= 0}$$

Recall that the loop invariant must be true before the loop executes, after every iteration of the loop, and after the loop terminates (though not necessarily during the loop body). Convince yourself that this loop invariant meets those conditions. We'll prove that it does soon.

Next we use the inference rule for while loops as a kind of template. It tells us all the places where we need to assert that the loop invariant holds: before the loop (line 4 below), at the start of the loop body (line 6), at the end of the loop body (line 9), and after the while statement (line 11). The pre-condition for the loop body (line 6) also includes the loop condition, c > 0.

```
1       //@ assert b > 0;
2   r = 0;
3   c = b;
4       //@ assert r == a * (b - c) && c >= 0;
5   while (c > 0) {
6           //@ assert r == a * (b - c) && c >= 0 && c > 0;
7       r = r + a;
8       c = c - 1;
9           //@ assert r == a * (b - c) && c >= 0;
10  }
11      //@ assert r == a * (b - c) && c >= 0 && c <= 0;
12      //@ assert r == a * (b - c) && c == 0;
13      //@ assert r == a * b;
```

The inference rule for while loops requires that we prove that the pre- and post-conditions are correct for the body of the loop. Once we've done that we will have backed the proof up from program post-condition to line 4. To prove that the body of the loop preserves the loop invari-

ant, we'll use the rules for assignment and sequencing.

The first step is to use the rule for assignment statements to derive a pre-condition for the last statement in the loop. Again using the rule as a template, we find all instances of the variable c in the post-condition and replace them with the expression c - 1 that appears on the right-hand side of the assignment statement.

```
1       //@ assert b > 0;
2   r = 0;
3   c = b;
4       //@ assert r == a * (b - c) && c >= 0;
5   while (c > 0) {
6           //@ assert r == a * (b - c) && c >= 0 && c > 0;
7       r = r + a;
8           //@ assert r == a * (b - (c-1)) && (c-1) >= 0;
9       c = c - 1;
10          //@ assert r == a * (b - c) && c >= 0;
11  }
12      //@ assert r == a * (b - c) && c >= 0 && c <= 0;
13      //@ assert r == a * (b - c) && c == 0;
14      //@ assert r == a * b;
```

We next calculate the pre-condition for the first statement in the loop. Starting with the assertion from line 8 above, we replace all occurrences of the variable r with the expression r + a.

```
1       //@ assert b > 0;
2   r = 0;
3   c = b;
4       //@ assert r == a * (b - c) && c >= 0;
5   while (c > 0) {
6           //@ assert r == a * (b - c) && c >= 0 && c > 0;
7           //@ assert (r+a) == a * (b - (c-1)) && (c-1) >= 0;
8       r = r + a;
9           //@ assert r == a * (b - (c-1)) && (c-1) >= 0;
10      c = c - 1;
11          //@ assert r == a * (b - c) && c >= 0;
12  }
13      //@ assert r == a * (b - c) && c >= 0 && c <= 0;
14      //@ assert r == a * (b - c) && c == 0;
15      //@ assert r == a * b;
```

Does the assertion in line 6 above imply the one in line 7? It does, but we need to do some arithmetic to show that. We introduce additional assertions to show our reasoning.

```
1     //@ assert b > 0;
2  r = 0;
3  c = b;
4     //@ assert r == a * (b - c) && c >= 0;
5  while (c > 0) {
6        //@ assert r == a * (b - c) && c >= 0 && c > 0;
7        //@ assert r == a * (b - c) && c >= 1;
8        //@ assert r == a * (b - c) && (c-1) >= 0;
9        //@ assert (r+a) == a * (b - c) + a && (c-1) >= 0;
10       //@ assert (r+a) == a * (b - c + 1) && (c-1) >= 0;
11       //@ assert (r+a) == a * (b - (c-1)) && (c-1) >= 0;
12    r = r + a;
13       //@ assert r == a * (b - (c-1)) && (c-1) >= 0;
14    c = c - 1;
15       //@ assert r == a * (b - c) && c >= 0;
16 }
17    //@ assert r == a * (b - c) && c >= 0 && c <= 0;
18    //@ assert r == a * (b - c) && c == 0;
19    //@ assert r == a * b;
```

Verify for yourself that each assertion implies the next in lines 6 through 11 above. (It may be easier to work backwards; that's what we did.)

Having proved that the body of the loop preserves the loop invariant, all that remains is to back the loop pre-condition up through the initial assignment statements. We replace each occurrence of the varaible c with the expression b.

```
      //@ assert b > 0;
   r = 0;
      //@ assert r == a * (b - b) && b >= 0;
   c = b;
      //@ assert r == a * (b - c) && c >= 0;
   while (c > 0) {
      ...
   }
      //@ assert r == a * (b - c) && c >= 0 && c <= 0;
      //@ assert r == a * (b - c) && c == 0;
      //@ assert r == a * b;
```

Then we calculate the pre-condition of the first statement in the program by replacing each occurence of the variable r with the expression 0.

```
    //@ assert b > 0;
    //@ assert 0 == a * (b - b) && b >= 0;
r = 0;
    //@ assert r == a * (b - b) && b >= 0;
c = b;
    //@ assert r == a * (b - c) && c >= 0;
while (c > 0) {
    ...
}
    //@ assert r == a * (b - c) && c >= 0 && c <= 0;
    //@ assert r == a * (b - c) && c == 0;
    //@ assert r == a * b;
```

Finally, we introduce additional assertions to show that the programs pre-condition implies the calculated pre-condition.

```
1     //@ assert b > 0;
2     //@ assert b >= 0;
3     //@ assert 0 == a * 0 && b >= 0;
4     //@ assert 0 == a * (b - b) && b >= 0;
5   r = 0;
6     //@ assert r == a * (b - b) && b >= 0;
7   c = b;
8     //@ assert r == a * (b - c) && c >= 0;
9   while (c > 0) {
10    ...
11  }
12    //@ assert r == a * (b - c) && c >= 0 && c <= 0;
13    //@ assert r == a * (b - c) && c == 0;
14    //@ assert r == a * b;
```

Now line 1 implies line 2, because if b is greater than zero then it is certainly greater than or equal to zero. Line 2 implies line 3, because we can always conjoin a tautology (i.e., "and on" an expression that is always true) without changing the meaning. Finally line 3 implies line 4 by arithmetic.

Putting all this together we get the following proof of partial correctness for the program:

```
    //@ assert b > 0;
    //@ assert b >= 0;
    //@ assert 0 == a * 0 && b >= 0;
    //@ assert 0 == a * (b - b) && b >= 0;
r = 0;
    //@ assert r == a * (b - b) && b >= 0;
c = b;
    //@ assert r == a * (b - c) && c >= 0;
while (c > 0) {
        //@ assert r == a * (b - c) && c >= 0 && c > 0;
        //@ assert r == a * (b - c) && c >= 1;
        //@ assert r == a * (b - c) && (c-1) >= 0;
        //@ assert (r+a) == a * (b - c) + a && (c-1) >= 0;
        //@ assert (r+a) == a * (b - c + 1) && (c-1) >= 0;
        //@ assert (r+a) == a * (b - (c-1)) && (c-1) >= 0;
    r = r + a;
        //@ assert r == a * (b - (c-1)) && (c-1) >= 0;
    c = c - 1;
        //@ assert r == a * (b - c) && c >= 0;
}
    //@ assert r == a * (b - c) && c >= 0 && c <= 0;
    //@ assert r == a * (b - c) && c == 0;
    //@ assert r == a * b;
```

To prove total correctness, we have to show termination. There is only one loop. The loop condition refers only to the variable c, which appears to be strictly decreasing. Since there is only one path through the loop, and that path includes the statement c = c - 1, we can assert that c is strictly decreasing. The loop condition bounds c from below, so the loop terminates.

Note that the process of proving partial correctness is a combination of applying rules and of manipulating expressions into the forms that rules require. Most of these steps can be automated. Probably the hardest step was the discovery of a loop invariant. JML provides syntax for specifying loop invariants and the bound function used to prove total correctness. When using tools to prove correctness it is handy to capture the loop invariant and the bound function and let the tool do the rest. Our original program specification with these given might look like the following:

```
    //@ assert b > 0;
r = 0;
c = b;
    /*@ maintaining c >= 0;
      @ maintaining r == a * (b - c);
      @ decreasing c; @*/
while (c > 0) {
    r = r + a;
    c = c - 1;
}
    //@ assert r == a * b;
```

The maintaining clauses give the pieces of the loop invariant; the expressions are conjoined together to get the loop invariant $P$ from the while inference rule. The decreasing clause gives the bound function—an integer-valued expression that must (i) get smaller with each iteration of the loop, and (ii) must be bounded by the loop.[1]

## References


[1] E. W. Dijkstra. Guarded commands, nondeterminancy, and the formal derivation of programs. *Comm. of the ACM*, 18(8):453–457, Aug 1975.

[2] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall, 1976.

[3] R. Floyd. Assigning meanings to programs. *Mathematical Aspects of Computer Science*, XIX: 19–32, 1967.

[4] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Third Edition*. The Java Series. Addison-Wesley, Boston, Mass., 2005.

[5] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. of the ACM*, 12(10): 576–583, Oct. 1969.

[6] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, P. Chalin, and D. M. Zimmerman. JML Reference Manual. Available from http://www.jmlspecs.org, May 2008.


[1] JML has no increasing clause. A strictly increasing bound function can be represented by giving its negation in a decreasing clause.