ProveIt: a C program correctness prover

W. D. Maurer

Department of Electrical Engineering and Computer Science

The George Washington University

Washington, DC 20052, USA

Phone: (1-202)994-5921

Fax: (1-202)994-0227

E-mail: maurer@seas.gwu.edu

Abstract

This paper describes a program known as ProveIt, which proves the correctness of C programs. Input to ProveIt consists of a C program with comments of a specific kind, which tell ProveIt what to prove (a single program can have several different proofs). Verification conditions are then generated and proved by using a knowledge base. Any programs for which the knowledge base is insufficient give rise, in the usual ways, to augmentation of the knowledge base. ProveIt is itself written in C++ and was debugged using CodeWarrior on the Macintosh.

Keywords

Correctness, validation, verification, verification condition generator, VCG

1 INTRODUCTION

ProveIt is a verification condition generator (VCG) and theorem prover for C programs, written in C++. Its chief advantages over similar programs are as follows:

- ProveIt uses the modification index method of verification condition generation (Maurer, 1977a and 1977b). This is easier to use than the conventional back substitution method, and, just as importantly, ties the generated verification conditions directly to the statements and assertions from which they arise. This greatly aids in the debugging of proofs of correctness.
- ProveIt can prove the correctness of a program containing goto statements. Indeed, it relies on Floyd's original inductive assertion method, which, as introduced in (Floyd, 1967), had nothing to do with structured programming. The result is that ProveIt can be used – as the author believes correctness techniques should be used – in proving the correctness of programs that are short, fast, and difficult to understand.

- ProveIt treats partial correctness, clean termination (the lack of unchecked errors – see (Sites, 1974)), and general termination (the lack of endless loops – see (Manna, 1970)) as parts of the same universal theory. (Many VCGs treat partial correctness only.) In addition, ProveIt contains a thorough (although optional) treatment of arithmetic overflow issues.

- ProveIt uses a knowledge base approach to the proof of validity of the generated verification conditions, rather than traditional searching methods with goals and subgoals. Human users are encouraged to enter their proofs into the knowledge base, which is then kept centrally on the Internet for searching by other users.

- ProveIt is written entirely by one author, who is also the author of over 30 papers on program correctness.

2 A SIMPLE EXAMPLE

As an unusually simple example of what ProveIt does, suppose that we give it the following input (the line numbers #1, #2, and the like are supplied by ProveIt):

```
#1      /* Multiplication Test for Version 0.11 */
#2      int k;
#3      /*: void f(int i, int j);
#4          theorem thm1;
#5          options nooverflow,noterminate;
#6          global {int k;}
#7          requires j > 0;
#8          produces k == i*j;
#9          updates k;
#10     */
#11     void f(int i, int j) {
#12 /*: prove thm1; */
#13         int n;
#14         k = 0;
#15         n = 0;
#16 loop: /*: here k == i*n, 0 <= n, n < j; */
#17         k = k+i;
#18         n = n+1;
#19         if (n < j) goto loop;
#20 /*: here k == i*j; */
#21         return;
#22     }
```

The following output is then produced by ProveIt (note the line numbers at the right, which show the origin of each line that ProveIt produces):

```
ProveIt™ Version 0.11, Tue Oct  8 14:52:15 1996
File name = INITIAL TEST
```

Verification conditions for function f:

CONDITION 1

V1	ia	if j > 0	Hypothesis	#7
V2	as	and k = 0	Hypothesis	#14
V3	as	and n = 0	Hypothesis	#15
V4	fa	then k = i*n		#16

V5	fa	and $0 \leq n$		#16
V6	fa	and $n < j$		#16

CONDITION 2

V7	ia	if $k = i*n$	Hypothesis	#16
V8	ia	and $0 \leq n$	Hypothesis	#16
V9	ia	and $n < j$	Hypothesis	#16
V10	as	and $k1 = k+i$	Hypothesis	#17
V11	as	and $n1 = n+1$	Hypothesis	#18
V12	cs	and $n1 < j$	Hypothesis	#19
V13	fa	then $k1 = i*n1$		#16
V14	fa	and $0 \leq n1$		#16
V15	fa	and $n1 < j$	V12	#16

CONDITION 3

V7	ia	if $k = i*n$	Hypothesis	#16
V8	ia	and $0 \leq n$	Hypothesis	#16
V9	ia	and $n < j$	Hypothesis	#16
V10	as	and $k1 = k+i$	Hypothesis	#17
V11	as	and $n1 = n+1$	Hypothesis	#18
V16	cs	and $n1 \geq j$	Hypothesis	#19
V17	fa	then $k1 = i*j$		#20

CONDITION 4

V18	ia	if $k = i*j$	Hypothesis	#20
V19	fa	then $k = i*j$	V18	#8

** ** *

Term type glossary:

as = assignment statement
cs = conditional statement
fa = final assertion
ia = initial assertion

End of program

3 VERIFICATION CONDITION GENERATION

We summarize here the basics of verification condition generation according to Floyd's inductive assertion method, as generalized by our own work over the years.

We will always assume that each of our programs has a single entry point. However, if there is more than one entry point, we simply introduce separate proofs for each entry point.

Our theory is applicable in the case of multiple exit points. However, in C, unless we use *setjmp* and *longjmp* (which are not treated in the present version of ProveIt), each function can have a maximum of two exit points: normal exit and halt exit (for which the program stops). Halt exit is normally brought about by calling a function with a halt exit, one of which is provided by the C library, and is called *exit(n)*. Some functions never halt, and some func-

tions always halt; but if a function F halts only some of the time, then a call to F is like a conditional branch, of the form ‘if (some condition) then halt.’

At any time during a run of a program, the statement S to be executed next is said to be *undefined* if executing S will give an unpredictable result. If S is undefined, we say that an *execution error* has been encountered. Sources of execution error include division by zero, subscripts out of range, arithmetic overflow, and calling a function under inappropriate conditions (such as trying to take the square root of a negative number).

An *assertion* is a Boolean expression, without side effects, involving the variables of the program and perhaps others as well. It may be more general than is allowed by the rules for forming Boolean expressions in the given language; in particular, it may be (and, in practice, is often) of the form ‘ $P(k)$ is true for all k from *min* through *max*.’

In order to formulate a statement of correctness about a function F , we first specify an *entry assertion* E (true when F starts), a *normal exit assertion* N (true when F exits normally), and a *halt exit assertion* H (true when F halts). If F never exits normally, but always halts, then N is *false*; if F never halts, then H is *false*. The statement of correctness of F now says that if E holds, and F is then executed, it will not loop endlessly; it will not encounter an execution error (as defined above); and that when it exits, it will either exit normally, in which case N will hold, or it will halt, in which case H will hold.

In order to prove this statement of correctness, we introduce key points within the program. At each key point K , we place intermediate assertions, which are supposed to hold whenever the program gets to K . In recent work (Maurer, 1996) we have shown that the ‘best’ choice of key points is almost always to choose the join points, that is, those points having indegree greater than 1, in the graph of the program. We will also refer to key points as *intermediate assertion points*.

In general, the *assertion points* of a function include its entry, normal exit, and halt exit points, with corresponding entry, normal exit, and halt exit assertions, as well as the key points, each with its corresponding intermediate assertion. Note that a program with several *return* statements has only one normal exit point; each *return* statement goes to the same place, and thus the normal exit assertions must always hold at all *return* statements. Similarly, a program that halts in more than one place has only one halt exit point.

A *verification path* of such a program is a path in its graph from one assertion point to another, with no assertion points in between. If the assertion points P_1 and P_2 occur just before statements S_1 and S_2 respectively, then a path Π from P_1 to P_2 includes S_1 but does not include S_2 (this is included in another path that starts at P_2). Associated with P_1 and P_2 are assertions A_1 (the *initial assertion* of this path) and A_2 (the *final assertion* of the path). Note that P_1 and P_2 may be the same, if the path goes around a loop.

For every verification path, we now define three levels of *verification conditions*. The first level condition of the path Π above says that if A_1 holds, and Π is now executed all the way to the end, then A_2 will hold. If the first level conditions of all verification paths hold, Floyd’s theorem says that the program is *partially correct*; that is, it is correct except for the possibility of endless looping and/or execution error, neither of which have been ruled out.

The proof of Floyd’s theorem is straightforward. Assuming no endless loops and no execution errors, we can use the assertion points of the function F to divide each execution sequence of F into subsequences which are verification paths. Let the assertion points be $P_0, P_1, P_2, \dots, P_K$, in order of the execution sequence, and with corresponding assertions $A_0, A_1, A_2, \dots, A_K$. If A_0 holds at P_0 , then A_1 holds at P_1 ; but if A_1 holds at P_1 , then A_2 holds at P_2 ; but if $\dots$ and so on, since the first level conditions of all these paths hold. Thus, if A_0 holds at P_0 , then A_K holds at P_K , which is precisely the statement of partial correctness of F .

3.1 Modification indices and guards

ProveIt uses the modification index (MX) method, introduced in (Maurer, 1977a and 1977b). Every time a variable with name n is modified, its new value is denoted by ni , where i is the

MX. These normally start at 1 and proceed in numerical order, with possible breaks in the sequence if ni is already a variable name, or if there is more than one variable with the name n . If n ends in a digit, the MX values start at a and proceed in alphabetical order. Forward accumulation is now used, according to Floyd's original method, rather than the back substitution of (Hoare, 1969). Current MX values are used at each point in the condition.

Provelt assumes that both assertions and program statements can be partial functions. Thus the assertion $x/y = z$ is partial; it is defined only when $y \neq 0$. We refer to $y \neq 0$ as the *guard* of $x/y = z$. If $x/y = z$ is an initial assertion of a path, then so is $y \neq 0$, and both may be assumed; if $x/y = z$ is a final assertion of a path, then so is $y \neq 0$, and both must be proved. Also, the assignment statement $z = x/y$ is partial; it is again defined only when $y \neq 0$. However, this time, the guards always have to be proved, if we are checking for execution errors. Suppose that $z = x/y$ represents the first modification of z within a verification path, so that z is here denoted by $z1$ (according to the MX method). Suppose further than x and y have not been modified yet. Then $z1 = x/y$ may be assumed, in this path, but before this, $y \neq 0$ must be proved. If y had been modified earlier in the path, so that its current MX is 2 (for example), then $z1 = x/y2$ may be assumed, but $y2 \neq 0$ must be proved.

3.2 Verification terms

A first level verification condition is composed of three kinds of terms:

- (1) Terms for the initial assertion of the path. No MX values are used.
- (2) Terms for the statements in the path. Every such term is derived from the given statement X by appending to each variable in X its current MX, or, for the left side of an assignment, its new MX. For a conditional statement, the given condition is inverted if the current path takes the false branch of the condition.
- (3) Terms for the final assertion of the path. This time, some of the variables may have been modified, so MX values are appended.

For a second level verification condition, we must introduce guards, as follows:

- (1) Each initial assertion having a guard is preceded by its guard, and the guard may be assumed, along with the initial assertion.
- (2) Each statement in the path which has a guard is preceded by its guard, and the guard (with MX values appended) must be proved.
- (3) Each final assertion having a guard is preceded by its guard, and the guard (with MX values appended) must be proved, along with the final assertion (also with MX values appended).

Note that, for first level conditions, the terms of types (1) and (2) are hypotheses, while the terms of type (3) are conclusions. The result is that all hypotheses precede all conclusions, and all hypotheses may thus be used in proving each individual conclusion. For second level conditions, the terms of type (2) are now a mixture of hypotheses (the terms themselves) and conclusions (their guards). It can now be shown that each conclusion in the condition has to be proved using *only* those terms in the condition which precede it.

3.3 Termination expressions and third level conditions

For third level conditions, we introduce *termination expressions* (see (Manna, 1970)). Suppose first that we have a program having a single loop, and a single intermediate assertion point P in that loop. A termination expression is then an integer expression, nonnegative at P , and always decreasing when the program goes from P around the loop and then back to P again. In the general case, we associate a termination expression with each intermediate assertion point in a program, and each of these must be an integer expression, nonnegative at its assertion point. Suppose then that we have a verification path P from point $L1$ to point $L2$.

Suppose that points L1 and L2 have associated termination expressions $t1$ and $t2$. Then $t2$ must be strictly less, at the end of P, than $t1$ was at the start of P.

We can derive third level terms from second level terms by introducing only two new term types. If t is a termination expression associated with the point L, then $t \geq 0$ becomes an intermediate assertion, with the same status as any other intermediate assertion at L. If L1, L2, $t1$, and $t2$ are as above, and $t2'$ is $t2$ with MX values appended as described above, then $t2' < t1$ must be proved at the end of the path. Note that $t1$, here, does *not* have MX values appended, because it is the value that $t1$ had when the given path was started.

3.4 Start variables

There are some programs whose proofs require additional variables. This is most common when these represent the starting values of variables that are later changed. Suppose, for example, that our program consists of the single statement $k = k+1$. What is its final assertion? It cannot be $k = k+1$, because k is not *equal* to $k+1$. It must be something like $k = k0+1$; but then what is $k0$? In Provel, $k0$ is called a *start variable*. It is declared in a *start* statement in Provel, as being equal to k when the program starts; and it may then be used in assertions.

3.5 The join point method and modification records

Much like a compiler, Provel must keep track of all variables in the source program. For each of these, a base record is constructed, and then, each time it is modified, a modification record is constructed, with a pointer to the base record. We can now show that each of these modification records may be associated with a unique MX, which can then be kept inside the record.

Note first that this is not true if the join point method is not used. For example, consider *if* (condition) $k = 5$; followed by $k = k+3$; with an assertion point immediately preceding the *if* statement. Suppose also that the assertion at that point involves k , and that the point just before $k = k+3$ is not an assertion point. In a verification path through $k = 5$, we would have $k1 = 5$ followed by $k2 = k1+3$. In a verification path through $k = k+3$ but not through $k = 5$, we would have $k1 = k+3$. Thus the left side of the assignment symbol in $k = k+3$ is here associated with two different MX values, 1 and 2. Note that the point just before $k = k+3$ is a join point, because there are two ways to get to it, one if the condition is true and the other if it is false. Hence the join point method cannot have been used here, because, if it were, this point would have had to have associated assertions.

If the join point method is used, however, then we can show, in general, that this situation cannot happen. If it did, then let X be the statement involving two different MX values. Consider the two paths involved, and remove the last few statements from each of them, after the statement X. The resulting paths must be the same for the last n statements, for some $n \geq 1$ (as large as possible), because at least the statement X is the same in both of them. The first of these n statements (call it Y) then must be a join point, because if there is only one statement Z that can precede Y, then Z must have preceded Y in both of our paths, contradicting our choice of n . But if Y is a join point, it must have assertions, by the join point method, and this contradicts the definition of verification paths.

3.6 Function calls

If a function f calls a function g with parameters, this is treated as a set of assignments of actual parameters to formal parameters, followed by a call to g . This follows from the fact that call (or pass) by value is used exclusively in C (except for array name and string name

parameters, which are currently not treated by ProveIt). The assignments are treated in the same way as any other assignment.

A call to g must involve a proof of g , and since there may be several proofs of g , one must be chosen (in an *apply* statement in ProveIt). The entry assertion of g (with appended MX values) now appears in the verification condition as a conclusion (this must be shown to hold, or else g will not necessarily work). At this point, new modification records are constructed for all variables changed by g . Such variables, in turn, have to be declared as such in the given proof of g , by an *updates* statement in ProveIt. What happens next depends on whether g has a normal or a halt exit; specifically:

- If g has only a normal exit, the normal exit assertion of g (with appended MX values) now appears in the verification condition as a hypothesis; we can assume these to hold in this path since g has made a normal exit. The path now continues with the next instruction after the call to g , or terminates if this instruction is a join point.
- If g has only a halt exit, the halt exit assertion appears, as above, instead of the normal exit assertion. This is then followed by the halt exit assertion of f , since f has now halted.
- If g has both normal and halt exit assertions, there are then two kinds of paths through the call to g , one for the normal exit (treated as above) and one for the halt exit (treated as above). Note that there will be only one halt exit path, but there may be one or more normal exit paths, because the call to g might be followed by a conditional statement.

3.7 Join point and unreachable statement determination

It is not immediately obvious, in some cases, how to determine join points, and how to determine unreachable statements.

Example 1: *do* $\{ \{s\} \{t\} \text{return}\}; \text{while } (c);$ – Here the *do* test (that is, the point just before *while* (c)) is a join point if and only if there are two or more *continues* in $\{s\}$ and $\{t\}$. Note that $\{s\}$ can have two *continues*, or $\{t\}$ can, or $\{s\}$ can have one and $\{t\}$ can have one, etc.

Example 2: *if* (c) $\{s\}; \text{else } \{t\}; u;$ – Here u is unreachable if $\{s\}$ is a *goto*, *return*, *break*, or *continue*, and so is $\{t\}$. It is usually (although not always) reachable otherwise.

Example 3: If most of the cases of a *switch* end in *return* or in error routine calls, the point following the *switch* is a join point if and only if at least two of the *cases* end in *break* (or one case ends in *break* and the last case falls through).

ProveIt has a method of determining join points and unreachable statements, involving the setup of three variables, one Boolean and two integer, that are associated with a statement s . They are:

Boolean *can_go_on* – This is true for s if s can go on to the next statement in sequence, either directly or through a *break*.

int *break_count* – This is the number of times that s can do a *break* which goes to whatever encloses s . (If s is compound, an internal *break* out of s does not count; it merely contributes to *can_go_on*.)

int *continue_count* – This is the number of times that s can do a *continue* which goes to whatever encloses s .

3.8 Assertions and assertion functions

An assertion, in the inductive assertion method, cannot have side effects, and thus cannot contain calls to functions in the usual programming sense. It can, however, refer to functions in the mathematical sense. Thus the assertion $\text{GCD}(I, J) = \text{GCD}(M, N)$ refers to GCD, which is defined as the greatest of the common divisors of two integers; and this definition is independent of any program that might be used to calculate GCD.

There is some difficulty in formulating appropriate terminology for ‘functions in the mathematical sense, as contrasted with programs that calculate them.’ We cannot call them simply functions; in C, a function is a program. We cannot call them mathematical functions; in C, a mathematical function is a program to calculate cosines, logarithms, or the like. We cannot even call them abstract functions, if we are familiar with the terminology of Ada 95 and Java, in which this term is used in a very specific sense. For lack of a better term, we refer to these in Provel as *assertion functions*, since references to them appear only inside assertions.

In general, the form of an assertion in Provel is the same as the form of an expression in C, with the following differences:

- Function calls are allowed; however, these are not calls to function subprograms, but rather to assertion functions as described above.
- The equal sign (=) is interpreted as if it were the double equals (==).
- All operators which invoke side effects are disallowed. These include the alternate assignments (+, *= and the like) and the operators ++ and -- (in both postfix and prefix forms), as well as the equal sign and calls to function subprograms, as noted above.

3.9 The key point specification problem

Given an intermediate assertion A in a program P written in a language L, an assertion language must specify the key point K in P with which to associate A; and the way in which this is done is highly dependent upon whether L encourages or discourages labels. Consider first a language such as Basic, in which labels are required. In this case, K will be just before some statement S which has a label (or line number) N. Therefore, to specify K, all we have to do is to specify N. This in turn means that, in a VCG for Basic, we can set up a proof specification C for P, which is separate from the program P. All assertions in C are designated either as entry, exit, or intermediate assertions, and for each intermediate assertion A, the corresponding label L is specified (and the same for termination expressions).

In Fortran IV, it is not necessary for every statement to have a label, but every key point must still have a label because otherwise it would be reachable only from the statement immediately preceding, contrary to the principles of the join point method; and proof specifications can therefore be set up separately in this case as well. The advantages of this separation should be obvious: a program can have several proofs, and each proof is set up in a separate file, which is then read by the VCG in conjunction with the single program file.

The situation is different in a modern language such as C (or Pascal, or Ada). Labels are allowed, but they are discouraged, less strongly so in C than in the others, but it is still highly unlikely that most key points in C programs will be labeled. We could introduce artificial labels, but these might be different for different proofs of a program P. This last statement may seem inconsistent with our method, in which the labels are precisely the join points, but consider, for example, a proof with an entry assertion which assures that large sections of P are never executed. In such a case, it should not be necessary for the prover to go through these sections and determine the join points, a determination which is not always obvious, as we have seen.

In Provel, we therefore denote intermediate assertions in a program by comments of a specific kind. Each such assertion appears in a program at its corresponding intermediate assertion point. In order to prove a program correct, then, we make a copy of the source code and insert these comments at various points. Specifically, a Provel comment is a C comment whose first nonblank character is a colon; and there are 16 kinds of Provel comments.

We distinguish between proof declarations and proof statements. *Proof declarations* specify a theorem to be proved; *proof statements* indicate how it is to be proved. If a function calls another function which is separately compiled and uses a proof of that function, it has to be accompanied by a proof declaration for that function. The Provel declaration types are as follows:

- *theorem tname*; – Here *tname* is the name of the theorem being declared.
- *options olist*; – Here *olist* is a list of options, separated by commas, which may include: *nooverflow* (we are not checking arithmetic overflow in this proof); *noruntime* (we are not checking for runtime errors); and *noterminate* (we are not checking for endless loops).
- *afunc adef*; – Here *adev* is a definition of an assertion function as described above.
- *global vlist*; – Here *vlist* is a list of global variables which are used in assertions about this function. (Note that these variables need not be changed, or even used, by statements of the given function.)
- *local vlist*; – Here *vlist* is a list of local variables used in entry, exit, and halt assertions
- *requires alist* – Here *alist* is the list of entry assertions for this function; these are, in other words, the requirements for this function in order for it to be correct.
- *always alist* – Here *alist* is a list of assertions which always hold throughout the function (or at least at every entry, exit, and intermediate assertion point).
- *start vlist* – Here *vlist* is a list of names of start variables (as defined above) for this function.
- *produces alist* – Here *alist* is a list of exit assertions for this function which hold at the normal exit; these are, in other words, the conditions that the function produces.
- *halt alist* – Here *alist* is a list of exit assertions for this function which hold at the halt exit. (We recall from the above discussion that functions in C can have only two exits,
- *updates vlist* – Here *vlist* is a list of names of variables except for local, nonstatic variables) that are updated by this function; that is, their values may be changed. When the function is called, each of these variables is given a new modification record.

The ProveIt statement types are as follows:

- *prove tname* – This occurs once at the start of the function, and it indicates which theorem about this function is being proved. Note that we can have proof declarations for several proofs, but only one is actually proved, in any given run of ProveIt.
- *here alist* – Here *alist* is a list of intermediate assertions which are supposed to hold at the given point in the function.
- *termination texpr* – Here *texpr* is a termination expression (as described above) which is associated with the given point in the function.
- *apply tname* – This is used when a function *f* calls a function *g*, and there may be several theorems about *g*; it tells which one to apply, in the proof of *f*.
- *extra dlist* – Here *dlist* is a list of declarations of extra variables, other than start variables, which are needed in the proof and used in intermediate assertions.

4 REFERENCES

- Floyd, R. W. (1967) Assigning meanings to programs. *Proc. Symposium on Applied Mathematics* **19** (*Mathematical Aspects of Computer Science*), American Mathematical Society, Providence, R. I., 19–32.
- Hoare, C. A. R. (1969) An axiomatic basis for computer programming. *Communications of the ACM*, **12**, **10**, 576–580, 583.
- Manna, Z. (1970) Termination of programs represented as interpreted graphs. *Proc. Spring Joint Computer Conference*, 83–89.
- Maurer, W. D. (1977a) The modification index method of generating verification conditions. *Proc. 15th Annual Southeast Regional Conference*, Biloxi, Miss., 426–440.
- Maurer, W. D. (1977b) A new method of generating verification conditions. *Proc. Conference on Information Sciences and Systems*, Johns Hopkins Univ., Baltimore, 128–133.
- Maurer, W. D. (1996) A minimization theorem for verification conditions. *Proc. 8th International Conference on Computing and Information*, Waterloo, Ont., Canada.

Sites, R. W. (1974) Proving that computer programs terminate cleanly. *Ph. D. Thesis, Computer Science Department, Stanford University.*

5 BIOGRAPHY

Ward Douglas Maurer received a bachelor's degree in Mathematics from the University of Chicago, and continues to apply the classical mathematics he learned there (based on sets and functions rather than on mathematical logic) to the development of the mathematical science of computing. He then received his Ph. D. in Mathematics from the University of California (Berkeley) in 1965 and has been Professor of Engineering and Applied Science at The George Washington University since 1973. He is codiscoverer of the division method of hash coding and discoverer of quadratic rehash. As a hobby, he writes and arranges satirical songs.