EXECUTABLE ASSERTIONS - AN AID TO RELIABLE SOFTWARE

S. H. Saith
General Research Corporation
P. O. Box 3587
Santa Barbara, California 93105

Abstract

Two preprocessors, one for FORTRAN and one for PASCAL, have been implemented to allow "executable assertions" to be added to the source code. These assertions make it possible to carry out certain static and dynamic checks on the semantics of a program. They can detect errors in input data and prevent error propagation. It is possible to include remedial instructions to compensate for detected errors and provide fault tolerance. The assertions can be applied to all the data types available in the languages. Executable assertions can also be used in a proof of correctness. This paper describes the syntax of the assertions, which contain first-order logical expressions. Examples are given of their use in a simple but practical example, the calculation of the time that a ballistic projectile travels.

Introduction

Logical assertions have been used in the proof that a computer program is correct for the last ten years.[1-3] Despite this fact, the procedures required to carry out a proof are considered to be difficult and beyond the abilities of many programmers. So much so that there is much current interest in using symbolic execution[4-6] rather than a proof to demonstrate that a program is correct. This paper presents another preliminary step towards the use of a formal proof procedure for the development of reliable software. This step is given the name executable assertions.

Executable assertions are statements that a programmer makes about the variables in a program that should always be true. The statements are translated into tests which check whether they are true. If they are not, an error message is printed or an error recovery procedure is invoked.

Executable assertions are a subset of the assertions that can be used in a correctness proof. It is intended that as a programmer becomes more knowledgeable about a program, the executable assertions will be expanded until they encompass the assertions required for a formal proof. Until that time, the executable assertions can provide a means for a programmer to use knowledge about program variables to aid in testing, integration, maintenance, and fault tolerance.

Form of Executable Assertions

Simple Assertions

Executable assertions have been implemented in preprocessors for the FORTRAN and PASCAL programming languages. In each case the form of a simple assertion is the same as the form of any logical expression in the language. That is, it is the same as any expression that can be used in an IF statement in the language. Some examples are:

**FORTRAN ASSERTIONS**

ASSERT (100 .GT. 0)
ASSERT (TIME .GT. 0.0)
ASSERT (HEIGHT .GT. 18000.0)

**PASCAL ASSERTIONS**

ASSERT 100 > 0;
ASSERT TIME > 0.0;
ASSERT HEIGHT > 18000.0;

The first example is obviously valid: 100 is always greater than 0. The other two are more realistic examples of assertions which are useful for protecting the range of variables. The second assertion states that the variable TIME must always contain a positive number. The third assertion states that the variable TIME must always be greater than 18000.0.

The previous examples could be improved by the use of protection on both sides of the range of the variable. After all, it is rarely true that a program would work correctly if given any positive number greater than 0.0 or greater than 18000.0. It is much more likely that an upper bound and a lower bound are present. For example, if, in addition, the variable TIME was not to become greater than 100 and the variable HEIGHT was not to become more than 20000, the examples could be written as:

**FORTRAN ASSERTIONS**

ASSERT (TIME .GT. 0.0 .AND. TIME .LE. 100.0)
ASSERT (HEIGHT .GT. 18000.0 .AND. HEIGHT .LE. 20000.0)

**PASCAL ASSERTIONS**

ASSERT (TIME > 0.0) AND (TIME <= 100.0);
ASSERT (HEIGHT > 18000.0) AND (HEIGHT <= 20000.0);

The ASSERT statement can be used anywhere within the executable portion of the program. It provides a protection against the program operating with data which is in error because of failure within the program itself, failure of an input/output device, or failure in another program.

During execution, when an assertion such as the preceding examples is detected as being false, an error message is printed giving the module name and the statement number of the assertion which is false. The program does not stop executing, since other information may still be obtained from the program, there may be subsequent error recovery procedures, or the assertion itself may be incorrect.
Qualified Assertions

Simple assertions are not well suited to test that arrays are set correctly. For asserting that an array is correct, assertions with qualifiers are used. Two types of qualifiers are available: the universal qualifier "for all" (\( \forall \)) and the existential qualifier "for some" (\( \exists \)).

The universal qualifier is used when the assertion is true for all the elements of the array. Some FORTRAN examples are:

```
FORTRAN ASSERTIONS

ASSERT (.ALL. I .IN. (1,100) (A(I) .EQ. 0))
ASSERT (.ALL. L .IN. (1,N-1) (B(L) .LE. B(L+1)))
ASSERT (.ALL. J .IN. (2,72) (CARD(J) .NE. BLANK))
```

Any time a qualifier is used it refers to a range of indices that are applied to the array. In the first example, the assertion states that all the elements from A(1) to A(100) of the array A are equal to 0. The part of the assertion which states

\( I\ .IN. (1,100) \)

means that I takes on the values 1, 2, ..., 100.

In the second example, the assertion states that for all elements from 1 to N-1 of the array B, each element is less than or equal to the succeeding element. This is a formal way of stating that an array B of length N is sorted in ascending order. The third example states that for all elements between 2 and 72 of the array CARD, none are equal to the value in BLANK.

The same examples in PASCAL would appear as:

```
PASCAL ASSERTIONS

ASSERT ALL I IN 1 TO 100 IS A[I] = 0;
ASSERT ALL L IN 1 TO N-1 IS B[L] <= B[L+1];
ASSERT ALL J IN 2 TO 72 IS CARD[J] <> BLANK;
```

The existential qualifier is used when the assertion is true for at least one element in the array. Some examples are:

```
FORTRAN ASSERTIONS

ASSERT (.SOME. I .IN. (1,80) (CARD(I) .NE. BLANK))
ASSERT (.SOME. J .IN. (1,N) (A(J) .EQ. SEARCH))
```

```
PASCAL ASSERTIONS

ASSERT SOME I IN 1 TO 80 IS CARD[I] <> BLANK;
ASSERT SOME J IN 1 TO N IS A[J] = SEARCH;
```

An expanded version of the ASSERT statement can provide an application-dependent response to a false assertion. With the normal form of the ASSERT statement, a false assertion will cause a message to be printed listing the module name and the statement number of the assertion which was violated. In addition, an ASSERT statement may contain a FAIL clause which designates the processing to take place in the event the assertion is false. Some examples of the use of FAIL clauses include: print an application-dependent message, take some corrective action, or designate an error-recovery procedure. The control flow of a complete ASSERT statement is shown in Fig. 1.

```
STATAEMENTS PRECEDING
ASSERT STATEMENT
.getState
STATAEMENTS FOLLOWING
ASSERT STATEMENT
```

```
Figure 1. Control Flow of ASSERT Statement
```

In FORTRAN, the FAIL portion of the assert statement is used to invoke a block of code. An example is:

```
ASSERT (X .GE. 1.0E-125) FAIL(UNDERFLOW)
```

In PASCAL, the FAIL block of code is part of the ASSERT statement. The previous example when written in PASCAL appears as:

```
ASSERT (X .GE. 1.0E-125) FAIL(UNDERFLOW)
```

In FORTRAN, the block of code named UNDERFLOW is invoked if \( X < 10^{-125} \). In such a case the block sets X to zero and returns to the code following the ASSERT statement. In FORTRAN the block of code which attempts to rectify the problems that would be caused by subsequent statements operating upon erroneous information is separated from the main program. Note that the block name can be longer than standard FORTRAN names.

In PASCAL, the FAIL block of code is part of the ASSERT statement. The previous example when written in PASCAL appears as:
ASSERT X >= 1.0E-125
FAIL
X := 0.0
END FAIL;

Note that the ASSERT statement is treated as a single PASCAL statement. It was written on several lines to improve readability. Of course, the FAIL blocks in either case can be expanded to contain many statements such as:

ASSERT X >= 1.0E-125
FAIL
X := 0.0;
WRITELN ('X HAS BEEN SET TO ZERO IN FFT MODULE')
END FAIL;

which would cause the message in quotes to be printed as well as setting X to 0 and continuing.

"Initial" and "Final" Assertions

Although the ASSERT statement may be used anywhere in the executable portion of a FORTRAN or PASCAL program, there are reasons for designating separate assertion statements at the entry and exit of a module of a program. Besides the mnemonic value of using keywords such as INITIAL and FINAL to represent assertions which test entry and exit conditions, the assertions which are so stated may be used eventually in a formal verification that the program meets its specification.

At the entry to a module, the data that is acted on by that module should be checked for reasonableness by an INITIAL assertion, as diagrammed in Fig. 2. If no assertion is made about a variable that comes into a module, the assumption is that no value of that variable will cause the statements in the module to fail.

The form of an INITIAL statement is the same as that of an ASSERT statement. Examples for FORTRAN and PASCAL are:

INITIAL (TIME .GT. 0.0 .AND. TIME .LT. 1000.0)
INITIAL (TIME > 0.0) AND (TIME < 1000.0);

At the exit from a module, the results calculated by the module are checked by a FINAL assertion. Each variable that contains output information should be checked by a FINAL assertion. If not checked at that point, the assumption is that any value will not cause failure in another module. Examples of FINAL assertions are:

FINAL (HEIGHT .GT. MINALT .AND. HEIGHT .LT. MAXALT)
FINAL (HEIGHT > MINALT) AND (HEIGHT < MAXALT);

Application

One of the difficulties with formal program verification has been its restriction in general to simple toy programs. Executable assertions can be applied without restriction.

Consider the following problem, depicted in Fig. 3, as an exercise in the application of executable assertions to a realistic problem.

Figure 3. Ballistic Problem

Given that a ballistic projectile is fired at time = 0 from an altitude of h₁ with an initial velocity v₀ and at an elevation angle φ above the horizon, what is the time required for it to land at an altitude h₂?

Using the ballistic equation

height = h₁ + v₀t - \frac{1}{2} gt²

where v₀ = v₀ sin φ

the solution is

\[ t = \frac{v₀ + \sqrt{v₀² - 2g(h₂ - h₁)}}{g} \]
which can be implemented in a simple FORTRAN or PASCAL program.

Consider what obvious errors are possible if the program is written as depicted in Fig. 4, but without the INITIAL, ASSERT, and FINAL assertions.

- A value of \( h_2 \) which is above the highest point in the "trajectory" will cause the square root of a negative number to be attempted.
- A value of \( \phi \) which is greater than 180° or less than 0° may cause a negative value of time.
- A negative value of \( v_0 \) may cause a negative value of time.
- If the output has a magnitude between 0.0 and 0.001 or greater than 9999.999 it will be printed incorrectly.

Observing this list of errors, the user might say, "Most of these will not happen." The values of \( h \) are chosen from 50 meters/second to 200 meters/second, and the values of \( v_0 \) range from 0.2 to 0.8 radians, the values of \( v_0 \) range from 50 meters/second to 200 meters/second, and the values of \( h_1 \) and \( h_2 \) range from 0 to 5000 meters in increments of 10 meters. The user might grant that a value of \( h_2 \) could be chosen erroneously.

Hence immediately after the READ statement, INITIAL assertions guarantee that the input data meets these specifications. These now protect the program from receiving erroneous data. If the assertions are not placed in the module, the program in effect states by default that all values from the smallest negative number to the largest positive number are appropriate for the problem.

Now assume that when it is pointed out that too large a value of \( h_2 \) can cause a negative square root, the user indicates that in such a case it is desirable to know the maximum possible height and the time to that height.

For this situation the FAIL block is used to re-calculate the height and print an appropriate message when DISC is negative.

As a check on the result the FINAL assertion is used. Here the value for the time is placed back in the height equation. The difference between the resulting height and \( h_2 \) must be very small.

Now if erroneous data is given, there is a hardware failure, or someone changes a value in the program, the FAIL block will notice when DISC is negative.

This work was supported by the Army Ballistic Missile Defense Systems Command under contract DASG60-76-C-0030.

References