A SOFTWARE QUALITY ASSURANCE EXPERIMENT*

J. P. Benson
S. H. Saib

General Research Corporation
Santa Barbara, California

ABSTRACT

An experiment was performed to evaluate the ability of executable assertions to detect programming errors in a real time program. Errors selected from the categories of computational errors, data handling errors, and logical errors were inserted in the program. Assertions were then written which detected these errors. While computational errors were easily detected, data handling and logical errors were more difficult to locate. New types of assertions will be required to protect against these errors.

Key Words: assertions, error categories

CR Categories: 4.6

INTRODUCTION

Over the past two years a number of techniques designed to eliminate errors in software have been implemented in a collection of programs called the Software Quality Laboratory (SQLAB).[1-4] An important goal of this effort has been the ability to analyze "realistic" programs. By *realistic* we mean programs which can execute on current computers with current compilers, use floating point arithmetic, incorporate data structures, be composed of multiple modules, and have a total size of perhaps several thousand statements.

In order to demonstrate SQLAB's ability to locate errors, a medium size program (~1000 statements) was selected. The program simulates the tracking of objects using a radar and embodies many of the characteristics of a complex software system including multitasking and data structures composed of queries and records.

The experiment described in this paper was designed to evaluate the use of assertions in a real time program. The experiment consisted of adding errors to the test program from a list of the most common software errors. A number of errors from a set of error categories were selected and introduced into the test program. (During the course of the experiment some errors already present in the program were also detected.) Executable assertions were written to detect the errors and the program was run to verify that the errors were actually detected. The results suggest that some of the assertions could have been made part of the variable definition statements of the programming language itself rather than separate statements. In addition, three new types of assertions which would be useful in error detection were identified: variable range assertions, approximate result assertions, and sequencing assertions.

ASSERTIONS

Assertions for software were first proposed by Floyd[5] as a method for proving the correctness of programs. Floyd's assertions were statements expressed as logical formulas in the first-order predicate calculus which specified the correct operation of the code. These assertions are combined with the statements of the program to generate logical formulas. If these formulas can be shown to be valid, the program is considered correct.

More recently, Stucki and Foshee[6] have used executable assertions to dynamically verify the execution of programs. Executable assertions are translated by a preprocessor into executable statements. When the logical expression in the assertion is false, an error message is printed along with the values of the variables named in the assertion. Assertions can also be used to indicate when the value of a variable exceeds its declared range, to report an invalid array subscript, to specify particular values that a variable may take on, and to report when the value of read-only parameters are changed by invoked procedures. SQLAB also uses executable assertions which are translated into executable code by a preprocessor so that they can be evaluated at run time. When the expression in an assertion is false, it is reported and a FAIL block can be executed. Code in the FAIL block can institute recovery procedures to either correct the cause of the error or reinitialize the software.

In addition to executable assertions, SQLAB also uses assertions to perform "static analysis" on software. Static analysis refers to consistency checks which can be performed without having to execute the program. UNITS assertions are used to specify the physical units of variables. Each expression in the program is then analyzed by SQLAB to verify that the units are used consistently.

*This work was supported by the Army Ballistic Missile Defense Systems Command under contract DASG60-76-C-0050.

INPUTS and OUTPUTS assertions state which variables (global to a subroutine or procedure) are read-only, write-only, or read/write. Using these assertions, checks on the consistency of data usage (data flow analysis) in a program can be performed by a static analyzer.

ASSERTIONS AND ERROR CATEGORIES

A number of software error studies[7-11] have shown that the following types of errors occur most often:

- Computation errors - Using the wrong equation, overflow or underflow, missing or extraneous computations
- Data handling errors - Subscript errors, failure to initialize a variable, referencing or updating the wrong variable
- Logic errors - Missing tests, incorrect tests, incorrect sequencing

Since these are the errors most often found in software, we feel it is important to develop assertions to identify them.

Computation errors can be detected using assertions which place bounds on the computation being performed or provide alternative computations which achieve the same result (e.g., sorting by HEAPSORT or by QUICKSORT). Computation bounds can be either fixed ranges or linear or higher order approximations to the result being computed. The assertion

```
RECVTIME < 2.0*RANGE/VLITE
```

is an example of a computational bounds assertion. In this case, a bound for the approximate time at which to expect a radar return (RECVTIME) is stated using only the object's range and the speed of light. The movement of the object during the radar pulse and other second order calculations were ignored.

Data handling errors can usually be detected by assertions which specify ranges, units, scale factors and other specific information concerning the type, and computer representation of variables. The following assertion which constrains the range of the variable BEAM is an example of a data handling assertion.

```
INITIAL (FIRSTBEAM <= BEAM) AND (BEAM <= MAXBEAM);
```

This kind of assertion could be added to a programming language and required when a variable is declared. Instead, BEAM could be declared as

```
BEAM : INTEGER [FIRSTBEAM..MAXBEAM];
```

which means that the variable BEAM takes on integer values from FIRSTBEAM to MAXBEAM.

This declaration does away with the need for the assertion if the compiler can use the declaration to compile run-time checks on the value of BEAM. For languages like FORTRAN, however, these checks must be added as executable assertions. In addition, the assertions are valuable to further limit the range of a declared variable in a part of the program.

Logic errors and sequencing errors are the most difficult to detect using assertions. The information for writing these assertions must be found in the program specifications or by considering in detail the problem that the program is to solve. For example, incorrect program sequencing can only be detected if the correct sequencing is described apart from the program.

A SAMPLE PROGRAM

The program that we used in our experiment, ORDGEN, constructs a schedule for the operation of a radar. It is written in V-PASCAL[12], a preprocessor language which is translated into PASCAL.

The schedule is expressed in terms of a time line. Commands are executed by the radar at certain points in time. These commands are: send a radar pulse, point the radar in a particular direction, and listen for a return from an object. There are certain constraints placed upon the generated schedule. These constraints are both physically and intuitively obvious. The radar cannot both send pulses and receive echoes at the same time, and it takes a finite amount of time in which to change the direction in which the radar is sending or receiving.

The schedule is constructed from a sequence of radar action requests. Each request specified one of four types of radar pulse: search, verify, special search, or track. Each of these pulse types has associated with it the length of time required to send the pulse and the length of time which the radar should listen for an expected return. Each request also includes the direction in which the radar should send the pulse (the beam number), the time at which the pulse should be transmitted, and the time between the transmit pulse and the expected return. This is a measure of the distance from the radar to an object.

ORDGEN takes as input a sequence of these radar action requests and constructs as output a schedule of radar commands which do not violate the constraints discussed previously.

ERROR SEEDING EXPERIMENT

The goal of the experiment was to discover if assertions could be written to detect typical errors from all the error categories. Assertions for detecting the types of errors previously discussed were added to ORDGEN. We also introduced into ORDGEN errors chosen from the categories of software errors determined to occur most often. The data handling, computation, and logic errors added to ORDGEN are shown in Table 1. ORDGEN was then run and the assertion error reports examined to determine which assertions (if any) detected the errors.

Error C1 (a computation error) was already present in ORDGEN. A reference was made to a constant specifying pulse length rather than to a constant specifying receive window length while scheduling a receive window. This error had not been detected in the output from ORDGEN even though we had been using the program for several months!

TABLE 1
ERRORS ADDED TO ORDGEN

COMPUTATION ERRORS

- C1: Using the wrong variable name in an equation
- C2: Leaving out a computation
- C3: Adding an unneeded computation

DATA HANDLING ERRORS

- D1: Referencing the wrong variable name
- D2: Using the wrong arithmetic operator
- D3: Not initializing a variable correctly

LOGIC ERRORS

- L1: Leaving out a test
- L2: Using the wrong relational operator in a test
- L3: Executing the wrong sequence of decisions

Error D1 (a data handling error) occurred in ORDGEN when the wrong variable was referenced. In ORDGEN, pulse length and receive-window length are specified for each type of radar pulse (search, verify, special search, and track) using the PASCAL data structure of enumerated type. An array of pulse lengths and an array of receive-window lengths are indexed by pulse types; e.g., RWINDOW[SEARCH] refers to the length of the search-pulse receive window.

Using the wrong variable name in a statement is a very difficult error to detect by executable assertions. In this case, the error was detected by associating the indices of each array with the values to be stored in the array, using an assertion such as the one following:

```
ASSERT (INDEX3 = SEARCH =>
        RTNTIME = BSDUR + SEARCHWINDOW)
   AND (INDEX3 = VERIFY =>
        RTNTIME = BSDUR + VERIFYWINDOW)
   AND (INDEX3 = SPECIAL SEARCH =>
        RTNTIME = BSDUR + SPSEARCHWINDOW)
   AND (INDEX3 = TRACK =>
        RTNTIME = BSDUR + TRACKWINDOW);
```

This assertion was able to detect the replacement of RWINDOW[INDEX3] by PDURAT[INDEX3] in the statement

```
RTNTIME := BSDUR + RWINDOW[INDEX3];
```

It would appear that errors such as C1 and D1 are more easily caught by static analysis than by executable assertions. Providing an assertion for every variable reference is, in effect, duplicating each statement. Strong type-matching requirements may be one method for locating these errors. For example, the variable RTNTIME should be calculated using variables whose values are associated only with receiving radar returns. If the constants PDURAT and RWINDOW were associated with the types TRANSMIT_TIME and RECV_TIME, respectively, then errors such as C1 and D1 could be caught by type checking.

Error D2 (using the wrong arithmetic operator) was introduced into ORDGEN by changing the statement

```
REQUEST.XMITTIME := CURRENTTIME + PINOM;
```

to

```
REQUEST.XMITTIME := CURRENTTIME - PINOM;
```

This error was detected by the existing executable assertion in VERGEN

```
ASSERT CURRENTTIME < REQUEST.XMITTIME;
```

which states that the time at which a radar transmit pulse is requested must be later than the current time.

In general, using the wrong operator in an expression can be detected by specifying variable ranges or stating approximate bounds on the results of computations.

Error D3 (incorrect initialization of a variable) can usually be detected by range checks if the range of the variable can be declared. PASCAL allows some range checking through subrange types. In ORDGEN, this error was introduced by initializing the variable SPOUT to -1 rather than 0. This was detected by the assertion

```
INITIAL 0 <= SPOUT;
```

While static analysis methods can be used to detect whether a variable has been initialized, this type of checking is expensive. A better solution would be to allow (or require) that a variable's initial value be stated when the variable is defined by a statement such as

```
SPOUT : INTEGER [0..MAXSPOUT] INITIAL 0;
```

which would specify that SPOUT is to take on the values from 0 to MAXSPOUT and is initialized to 0. Verifying that the initial value is correct should still be done by executable assertions in the routines that use the variable.

Error C2 (leaving out a computation) was introduced into ORDGEN by leaving out a statement. This error was detected by the assertion

```
ASSERT RQUEST.BEAMPOS = TOTDSREC.OTBEAM;
```

In general, this kind of error can be detected by assertions which describe the computation that a routine performs or set bounds on the result of a computation. In the simplest case (as in the above example) the assertion may merely verify the action of a statement.

Error C3 (an extraneous computation) was another error that existed in ORDGEN but was not discovered until the experiment. The expression

```
+TOTDSREC.OTVCL*PITRK
```

which should have been deleted from the second line of

```
RQUEST.RECVTIME := ROUND (2*(OTRNGE + OTVCL*PITRK
                   + TOTDSREC.OTVCL*PITRK)/VLITE
                   - RWINDOW[TRACK]/4);
```

was mistakenly left in when this statement was corrected for another error. This error results in the calculation of an incorrect range. The error was detected by the assertion

```
ASSERT RQUEST.RECVTIME = ROUND (
       (2.0*OTDSREC.OTRNGE/VLITE - TRACKWINDOW/2)
       -(2.0*OTDSREC.OTVCL/VLITE - TRACKWINDOW/2));
```

which is just an alternative way of expressing the previous computation. As with the assertions for errors D1, C1, and C3, this assertion checks the result of a statement by duplicating (or equivalently expressing)(the expression in the statement.

Error L1 (leaving out a test) was implemented in ORDGEN by leaving out a decision. The effect of this error was that more than the maximum allowed number of search pulses were left outstanding (had not had their returns processed). This eventually caused the radar orders buffer to overflow. The assertion on the range of the number of outstanding search pulses

```
    ASSERT (0 <= SPOUT) AND (SPOUT <= MAXSPOUT);
```

detected this error.

Missing tests can only be discovered if the requirements for the program include these tests. In the above case, the redundant specification of the range of the variable SPOUT (which counts the number of outstanding search pulses) allowed the error to be detected.

Error L2 (using the wrong relational operator in a test) is a common error which is not easily detected. This was another error which was already present in ORDGEN and was not detected until the experiment. The incorrect test was part of a REPEAT...UNTIL construct. Instead of the correct test

```
        UNTIL NEXT > NUMBER - 1;
```

the test

```
        UNTIL NEXT >= NUMBER - 1;
```

was substituted. This resulted in the final entry in the radar activities queue not being processed.

The assertion to catch this error used an auxiliary variable (a variable not used elsewhere in ORDGEN) to count the number of radar requests processed. This number is then compared with the number of requests in the queue by the assertion

```
        ASSERT PROCESSED = NUMBER;
```

Error L3 (executing the wrong sequence of decisions) is another error that was already present in ORDGEN that was not detected by testing. Due to the misplacement in an IF..ELSE..ENDIF control structure, the statement

```
          PUT2 (REQUEST);
```

which returns an unprocessed request to the radar request queue, could be executed twice for the same queue entry. This results in duplicate entries in the queue.

This error can be detected by specifying the conditions that must be true whenever the ELSE clause of an IF statement is executed. This error points out a weakness in the ELSE construct, it allows a statement or block of statements to be executed whatever conditions are true. If decision statements in the program are executed out of their intended order, the conditions which were assumed to hold when the ELSE clause was executed may no longer be true. For this reason it is recommended that assertions be placed after all ELSE statements. These assertions should specify all conditions that must be true when the ELSE is executed.

For example, the following assertion was placed in ORDGEN in order to catch error L3. (The sequence of decisions is repeated here to show the source of the assertion.)

```
IF XMITTIME < EFRAM THEN
      .
      .
      .
IF IR + BSDUR + RWINDOW[INDEX3] < EFRAM THEN
      .
      .
      .
IF IX + BSDUR + PDURAT[INDEX3] > XLATE THEN
      .
      .
      .
IF RFIRST + RECVTIME + BSDUR + RWINDOW[INDEX3]
          < EFRAM THEN
      .
      .
      .
ELSE
   ASSERT (XMITTIME < EFRAM)
      AND (IR + BSDUR + RWINDOW[INDEX3] < EFRAM)
      AND (IX + BSDUR + PDURAT [INDEX3] > XLATE)
      AND (RFIRST + RECVTIME + BSDUR
                 + RWINDOW[INDEX3] >= EFRAM)
      .
      .
      .
ENDIF;
```

RESULTS

The experiment showed that executable assertions could be written to detect the most common types of programming errors. They appear to be most valuable in catching computational errors. Many computational errors can be found by specifying variable ranges and by stating approximate bounds on the results of computations. These types of assertions are so valuable that they should probably be considered as a separate type of assertion from the normal logical assertions. Assertions can also be used for detecting some data handling and logic errors. However, other types of analysis such as static analysis and sequence analysis seem more promising for these types of errors.

The addition of assertions to the program increased its compilation time by 56 percent, its execution time by 12 percent, and its storage requirements by 13.5 percent. This data has been substantiated in several other programs, including SQLAB itself.

As a result of the difficulties in writing assertions to detect errors in program logic and incorrect sequencing, we are developing two extensions to executable assertions to detect these types of errors. In one, we use a finite state machine model to indicate correct sequencing in a program, and in the other we use the idea of "auxiliary variables" proposed by Owiki and Gries[13] to retain information about the program in addition to that stored in the program's variables. The ability of these types of assertions to detect errors will be evaluated in future experiments.

REFERENCES

1. J. P. Benson and R. A. Melton, "A Laboratory for the Development and Evaluation of BMD Software Quality Enhancement Techniques," Proceedings of the Second International Conference on Software Engineering, IEEE Catalog No. 76CH1125-4C, IEEE Computer Society, Long Beach, California, October 1976, pp. 106-109.

2. D. M. Andrews and J. P. Benson, Advanced Software Quality Assurance, Software Quality Laboratory Users Guide, General Research Corporation CR-4-770, May 1978.

3. S. H. Saib et al., Advanced Software Quality Assurance, Final Report, General Research Corporation CR-6-720, March 1977.

4. S. H. Saib et al., Advanced Software Quality Assurance, Final Report, General Research Corporation CR-3-770, March 1978.

5. R. W. Floyd, "Assigning Meanings to Programs," in Proceedings of a Symposium in Applied Mathematics, J. T. Schwartz, Ed., Vol. 19, "Mathematical Aspects of Computer Science," American Mathematical Society, 1967, pp. 19-32.

6. L. G. Stucki and G. L. Foshee, "New Assertion Concepts for Self-Metric Software Validation," Proceedings International Conference on Reliable Software, IEEE Catalog No. 75CH0940-7CSR IEEE Computer Society, Long Beach, April 1975, pp. 59-71.

7. M. J. Fries, Software Error Data Acquisition, Boeing Aerospace Company RADC-TR-77-130, April 1977.

8. A. B. Endres, "An Analysis of Errors and Their Causes in System Programs," IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, June 1975, pp. 140-149.

9. T. A. Thayer, et al., Software Reliability Study, TRW Defense & Space Systems Group 76-2266.1.9-5, August 1976.

10. R. W. Motley and W. D. Brooks, Statistical Prediction of Programming Errors, IBM Corp. Federal Systems Division RADC-TR-77-175, May 1977.

11. J. A. Dana and J. D. Blizzard, Verification and Validation for Terminal Defense Program Software: The Development of a Software Error Theory to Classify and Detect Software Errors, Logicon HR-74012, May 31, 1974.

12. N. B. Brooks and E. N. Kostruba, Verifiable PASCAL Users Manual, General Research Corporation, February 1978.

13. S. Owiki and D. Gries, "Verifying Properties of Parallel Programs: An Axiomatic Approach," Communications of the ACM, May 1976, pp. 279-285.