Simple dynamic assertions for interactive program validation

by CHRISTER HULTEN
Syslab, University of Stockholm
Stockholm, Sweden

ABSTRACT

It is well known that more than 50% of software life cycle costs are caused by maintenance activities: testing, debugging, modification, regression testing, and documentation updating. Therefore the importance of the validation and verification process in software development cannot be overstated. An interesting technique introduced by Stucki is to instrument a program with dynamic assertions. The assertions, which are logical expressions regarding program variables, are entered into the program as comments, after which a preprocessor generates and inserts the code for dynamically checking the validity of these assertions. A number of papers describe more or less sophisticated and complicated ways of using dynamic assertions in test systems. The aim of this paper is not to analyze and compare these approaches with each other or with our proposal, but rather to convey the advantages of a simple, user-friendly system based on dynamic assertions for expressing constraints, transactions, and transition constraints.
INTRODUCTION

It is well known that over 50% of the software life cycle costs are caused by maintenance activities: testing, debugging, modification, regression testing, and documentation updating. Therefore the importance of the validation and verification process in software development cannot be overstated. A survey of validation, verification, and testing techniques for computer software can be found in the work by Adrion and colleagues. 1

Program test methods can be classified into static and dynamic methods. A static program test method does not involve executing the target program, but rather executing an analysis program that examines the source level program and tries to find errors or anomalies in the target program. Typical static methods are data flow and control flow analysis, compiler syntax and type checking, and symbolic execution and formal verification (proof techniques). Dynamic program test methods, on the other hand, do involve execution of the target program, albeit sometimes in a modified form. Typical dynamic methods are “traditional” program test methods with various test data generation techniques, 7 instrumentation and measurement techniques, and finally dynamic assertion techniques. This paper is concerned with a system for simple use of dynamic assertions.

DYNAMIC ASSERTIONS

An interesting technique introduced by Stucki 9 is to instrument a program with dynamic assertions. The assertions, which are logical expressions regarding program variables, are entered into the program as comments, after which a preprocessor generates and inserts the code for dynamically checking the validity of these assertions. A number of papers describe more or less sophisticated and complicated ways of using dynamic assertions in test systems. 2,5,6,8,9 The aim of this paper is not to analyze and compare these approaches with each other or with our proposal, but rather to convey the advantages of a simple, user-friendly system based on dynamic assertions.

There are several important benefits in using dynamic assertions:

1. The assertions should be invented at program design time, because it means that the programmer is encouraged to think in detail about what assertion is valid at a particular time in the execution of a program as well as about what invariants are valid throughout the execution of the program. This in itself will catch a substantial amount of errors that would otherwise turn up much later in the software life cycle, with corresponding higher costs for error detection and correction. 3

2. A program in a conventional programming language is a procedural description of how to achieve some state of affairs. If it is possible to describe this state of affairs in a declarative way, which is a complementary way of looking at the problem, then many algorithmic errors may be detected not only at run time but also at design time, since the programmer will think about the problem in two complementary ways. Of course there is a probability of introducing errors when making the assertions; but at least this does not introduce errors into the program proper (and of course it is hoped that these errors will be discovered). Since the procedural and the declarative descriptions are very different, I believe it unlikely that the same errors would be made in both descriptions—that a program error would be undetected because of a corresponding assertion error. Moreover, if an error is made in an assertion, it may well reflect the fact that the problem is not well understood. A discrepancy between the declarative and procedural representations should be thought of as if the error were equally likely to be in either of the representations.

3. If appropriate dynamic assertions are inserted in a program in strategic places, debugging is greatly facilitated, and errors can be pinpointed more quickly. However, it requires that software for assertion facilities either be integrated with the compiler or be structured into a preprocessor and a postprocessor. The reason for this is the need to transform the regular error messages from the compiler and run-time system into messages regarding the true source program—i.e., the source code, including the assertions. In a more sophisticated system, integration is carried further, so that other items of software (e.g., screen editors and debuggers) are given intelligence in terms of the assertion subsystem. An assertion subsystem can be made rather independently of a compiler in a good programming environment and fits in nicely with a good interactive system.

4. One of the most important activities in most program testing methods is the construction of test case results: For every test case—sets of input data to the program—a set of output conditions must be described, and must be described in advance, so that at least a manual check of a test case can be made. 7 It is well known that programmers frequently resent and fail to describe test case result construction. I believe that one reason for this is that the test case result descriptions are not part of the final product (the production program) but only a tedious component of the “destructive” 7 testing process. In this context we advocate the use of assertions as a natural
I will now outline an assertion system that I believe is simple enough to be accepted and used by programmers and yet powerful enough to achieve the advantages mentioned above. For presentation purposes the examples will be in PASCAL. The environment in which we place an assertion system is a Berkeley Unix/C system, an environment that highly facilitates implementation of such systems. It is assumed that there exists a symbolic debug system that can be interfaced to the assertion system.

Constraints

Since it is very important to keep the number of concepts low, only two are used here: constraints and assertions. A constraint expression is a Boolean expression (it can be evaluated to True or False) in the regular programming language style. It may contain Boolean function calls, which in turn may contain other function or procedure calls. This, of course, allows complex evaluations to be made. No assumption may be made with regard to the order of evaluation of the constraint expression. This means that if any assignments are made to program variables due to function or procedure calls executed when evaluating a constraint expression, care must be taken to ensure that the constraint expression evaluation order is insignificant. The reason for allowing assignment operations at all in constraint expressions (indirectly) is that it may be necessary to set up some conditions before performing the evaluation. An example of this occurs when a constraint is concerned with the consistency between an external database and program variables.

A constraint has the following structure:

\[ E:C:V \]

where

- \( E \) denotes Enforcement condition
- \( C \) denotes Constraint expression
- \( V \) denotes Violation action

The constraint semantics are as follows: If \( E \) is evaluated to True, then \( C \) (the actual constraint condition) is evaluated. If \( C \) is evaluated to False—i.e., the constraint condition is violated—then the statement \( V \), which of course can be a compound statement, is executed. The assertion system additionally reports the violation and, when appropriate, passes control to the debug system. The condition \( E \) is used for controlling the individual evaluation of a constraint expression. A constraint does not have to have all these three components. The other valid combinations are as follows:

- \( C \)—this is probably the most common variation. It means that only a constraint condition is specified, and if it is found to be False, the violation is reported and the execution aborted.
- \( E:C \)—this means: If \( E \) evaluate \( C \). If \( C \) is False, report violation and abort.
- \( C:V \)—this means: Always evaluate \( C \). If \( C \) is False, execute \( V \), report violation, and abort conditionally.

A constraint declaration has the form:

```
CONSTRAINTS E1:C1:V1;
E2:C2:V2;
   ...
En:Cn:Vn;
ENDCONSTRAINTS;
```

A CONSTRAINTS declaration can be placed anywhere in the program where a variable declaration is legal and where the scope of the constraint evaluation is the same as the scope of variables declared in that block or other entity. Furthermore, the enforcement of constraints will be in effect on that block level as well as on inner block levels. The constraints in a CONSTRAINTS declaration are monitored; i.e., the variables referred to in a constraint expression are checked after each explicit or implicit assignment operation. Note that, since functions and procedures can be used in constraint expressions, it is not permitted to have side effects in functions/procedures on any level in a constraint expression—i.e., assignments to variables on the same level as, or global to, the CONSTRAINTS declaration. The reason for this is, of course, that when such a variable is updated and checked, the side effect will cause successive checks and con-
sequently result in infinite recursion. This restriction, however, is not very limiting, since virtually all checking procedures will only read global variables, if any. Again, as in many cases in programming, side effects and global variables turn out to be harmful.

A CONSTRAINTS declaration is intended to represent conditions that are relatively independent of the individual statements in the program. The system therefore monitors every assignment operation where a variable involved in a constraint expression is changed. In many cases the program variables will be in an inconsistent state for a short while over several assignment operations—e.g., while transferring money between two bank accounts. There is then a need for a primitive to define transactions within which checking of the constraints is not meaningful.

Furthermore, since the monitoring of many assignment operations is very demanding from a performance point of view, it should be possible to turn off constraint checking in certain parts of the program. There is also a need for transforming a program with constraints into an efficient version without any constraint system overhead at all. There are two pairs of primitives for turning the assignment monitoring off and on. The first pair is concerned with excluding certain parts of the program from dynamic assignment checking, the second with eliminating constraints completely from the object program.

The first primitive is NOCONSTRAIN, which, when executed in the program, turns off all constraint checking until a CONSTRAINT primitive is encountered. The executing program is either in the NOCONSTRAIN mode or the CONSTRAINT mode. The default is the CONSTRAINT mode. Transactions are formed between NOCONSTRAIN-CONSTRAINT pairs. The third primitive is #NOCONSTRAIN, a preprocessor command, which statically turns off the preprocessor generation of constraint-checking code. #CONSTRAIN turns it on again. The #-commands are performed as the preprocessor scans the source program lexically from start to end.

**Assertions**

As seen, constraints are suitable for monitoring updates of variables throughout a program. If particular conditions must hold at a particular point in the execution of the program, it is useful to have another type of primitive to assert that these conditions hold. Typical conditions of this are (1) entry/exit conditions in procedures and functions and (2) assertions about a database state or loop invariants. The primitive for this purpose is the ASSERTION statement. An ASSERTION statement has the following structure:

```
ASSERT E1:C1:V1;
E2:C2:V2;
E3:C3:V3;
ENDASSERT;
```

Each Ei:Ci:Vi (i = 1, 2, 3, ...) is a constraint with the same semantics as before, but evaluation is performed only when the ASSERT statement is executed. As with CONSTRAINTS, there are primitives for disabling assertion statements and removing assertion statements. They are: NOASSERT, ASSERT, #NOASSERT, and #ASSERT.

**Transition Constraints**

In database literature it is commonly considered desirable to be able to describe a type of constraint called transition constraints. A transition constraint is a constraint involving the values of a variable, before and after an update; e.g., a salary must not be increased by more than 10%.

From an assertion system point of view, there are two ways of achieving this. The first way is simple to program the recording of the preconditions manually; e.g.,

```
VAR x:INTEGER;
FUNCTION rec_precond:INTEGER; ................. ;
FUNCTION dyn_check (i:INTEGER) :BOOLEAN; ...... ;
BEGIN
  x: = rec_precond;
  S1;
  S2;
  S3;
  dyn_check (x)
END;
```

where S1, S2, and S3 are PASCAL statements.

Another way of handling transition constraints is by the use of a system-defined function OLD(x). The difference is here that the system takes care of the administration of the prior value of a variable. The resulting type of OLD(x) is the same as that of x. The value of OLD(x) is the next latest value assigned to x. Using OLD(x), however, may be very costly, since code will be generated for saving the previous value of the variable and distributed all over the program.

An example: Assume that x may not increase by more than 10%:

```
PROGRAM maxten;
VAR x,y,z:INTEGER;
b:BOOLEAN;
BEGIN
  IF b THEN x: = x + y ELSE x: = x + z
  ASSERT x < OLD(x)*1.1;
END.
```
The manual method is more appropriate for complex transition constraints, whereas the OLD(x) function is possible only in simple cases.

SOME IMPLEMENTATION ASPECTS

An assertion system like the one proposed must consist of a preprocessor and a postprocessor. The main function of the postprocessor is to be a bridge between the high-level source program (the source program with assertions and constraints) and the conventional source program (the program generated by the preprocessor). The main problem, of course, is that when the compiler, the run-time system, and the operating system detect error conditions or exceptions, they return information about the generated program, not about the high-level source program. This means that it should be possible to trap run-time errors—e.g., divide by zero—so that the preprocessor can generate code for interrupt routines and communicate to the postprocessor. Further, when an error is detected, the high-level source code should be loaded into an editor that pinpoints the error, if possible.

If the restriction that no global variables may be used indirectly within constraint expressions is enforced, it enables a much less complicated preprocessor to be implemented (with a corresponding reduction in preprocessing costs). At the same time, the required run-time resources can be reduced. However, since I want to encourage usage of dynamic assertions, I feel that this restriction is too limiting.

SUMMARY AND CONCLUSIONS

This paper has attempted to point out some benefits of using a simple dynamic assertion system for typical industrial program production. I strongly believe that an approach like the one suggested has a high potential for cutting software development and maintenance costs. However, a study of the effects of using such a system in a commercial environment is necessary for assessing the approach in a quantitative way.

A drawback of dynamic checking techniques is that the run-time cost (in terms of time and space) of a program is increased, sometimes to an unacceptable degree. An advantage in this approach is that you can, to a large extent, control the amount of run-time checking and not pay for more run-time checking than you want.

It is advisable to have different levels of run-time checking in different phases of a program’s life cycle. During the initial program testing you would preferably have maximum constraint and assertion checking. In preliminary production runs perhaps some of the checking would be reduced, and in heavy production runs probably only the main assertions would be activated. In time/space critical applications perhaps the run-time checking would be eliminated altogether. After program modification it is advantageous to turn on all checking again.

As mentioned, it is easy to incorporate new and advanced static program analysis methods like program control flow analysis and formal proof methods. Increased use of such methods should of course be matched by a corresponding reduction in run-time checking.

If the target language is not strongly typed, a pre- and postprocessor system for dynamic assertions can help achieve the reliability that characterizes strongly typed languages. Of course, using an assertion system well results in reliability far beyond type checking, but at the expense of more run-time checking.

Finally, I believe that an assertion system must be very simple to be used at all by industrial programmers. This paper has attempted to show what a simple system with controllable overhead can look like and to point out some of its properties.

REFERENCES