Assertions in programming:
from scientific theory to engineering practice

Tony Hoare
Microsoft Research Ltd., Cambridge

Belfast April 2002

I first became interested in assertions when I took up my post as Professor of Computer Science at the Queen's University Belfast in 1968. I welcomed them first as a means of applying objective scientific judgement to the design and evaluation of computer programming languages. My subsequent academic research career was inspired by an ideal that assertions would contribute to the avoidance of programming error by means of mathematical proof.

At the present day, assertions are widely used in program development practice, as I have found from a recent survey among software development managers in Microsoft. The main role of assertions is as test oracles, to detect programming errors as close as possible to their place of occurrence. In this talk I will describe a number of other ways in which assertions are found useful, and may become even more useful in future. ~~All of them~~ fall far short of the hard ideals of mathematical proof.

The practical use of assertions in engineering will always

scientific

An assertion

- is a Boolean expression
- written in the middle of a program
- which is always true
- whenever control reaches that point.

- At least, that's the intention

An assertion in its most familiar form is a Boolean expression that is written at any point in the program text. It can in principle or in practice be evaluated by the computer, whenever control reaches that point in the program. If an assertion ever evaluates to false, the program is incorrect. But if it always evaluates to true, then at least the relevant program defect has never been detected. But best of all, if it can be proved that the assertion will always evaluate to true on every possible execution, then the program is proved to be correct, at least with respect to that assertion.

Program verification

- Alan Turing (1950)
  On Checking a Large Routine
- John McCarthy (1963)
  A Basis for a Mathematical Theory of Computation
- Robert Floyd (1967)
  Assigning Meanings to Programs
- Edsger W. Dijkstra (1968)
  A Constructive Approach to the Problem of Program Correctness.

An understanding of the role of assertions in the checking of large computer programs goes back to Alan Turing in 1950. The idea of automatic theorem proving to guarantee the correctness of programs goes back to John McCarthy in 1963. The idea of assigning a meaning to a programming notation by specifying the logic of correctness proofs goes back to Bob Floyd in 1967. And the idea of writing the assertions even before writing the program was propounded by Edsger Dijkstra in 1968. Dijkstra's insight has been the inspiration of much of the subsequent programming research conducted in University computing departments throughout the world.

Prospect

Recommendation

5

What's wrong with Test?

- Testing can only show the presence of bugs.
- Never their absence (Dijkstra)

- But test is complementary to specification, reasoning and proof.
- It is fundamental to both Science and Engineering.

a very inferior substitute for proof

Of course, everyone knows that programs today are not proved correct; they are only tested with a greater or lesser thoroughness. As a researcher, I used to deplore this reliance on testing as inefficient and ineffective. I would quote with approval the famous dictum of Dijkstra, that program testing can prove the existence of program bugs, but never their absence.

But this has been a mistake. I now believe that testing is a valuable and necessary complement to specification, reasoning and proof of programs. And assertions are just as important for testing as they are for mathematical theory. I would like to explain my change of mind by an analogy with the role of testing in other branches of science and engineering.

the

that underlies proof

(2)

Experiment in Science.

- A scientific theory has independent credibility
  - based on *a priori* reasoning
- It is subjected to rigorous test,
  - designed to refute it (Karl Popper)
- If it passes all tests, it is accepted.

According to the teaching of Karl Popper, the testing of scientific hypotheses by rigorous experiment is essential to the progress of Science. A scientific advance starts with a theory that has some a priori grounds for credibility, for example, that a force has an effect that is inversely proportional to the square of the distance at which it acts. A new theory that applies such a principle to a new phenomenon is subjected to a battery of tests that have been specifically designed, not to support the theory, but to refute it. If the theory passes all the tests, it is accepted and used, perhaps to help in the formulation and test of further and more advanced theories.

Extending this analogy to computer software, we can see clearly why program testing is in practice such a good assurance of the reliability of software. A competent programmer always has a prior understanding, perhaps quite intuitive, of the reasons why the program is going to work. If this hypothesis survives rigorous testing regime, the software has proved itself worthy of delivery to a customer. If a few small changes are needed in the program, they are quickly made – unfortunate, but that happens to scientific theories too. In Microsoft, every project has assigned to it a team of testers, recruited specially for their skill as experimental scientists; they constitute about a half of the entire program development workforce.

a

kind of detailed adjustment is often made

(3)

Test in Engineering

- Analogy: engine on a test bench
- Instrumented by probes at internal interfaces
- To test tolerances continuously
- And avoid test to destruction
- Opportunity to improve quality by tightening the tolerances

My second is) ~~I turn now to the~~ analogy with other branches of engineering, where rigorous product test is an essential prerequisite before shipping a new or improved product. For example, in the development of a new aero jet engine, an early working model is installed on an engineering test bench for exhaustive trials. This model engine will first be thoroughly instrumented by insertion of test probes at every accessible internal interface. A rigorous test schedule is designed to exercise the engine at all the extremes of its intended operating range. By continuously checking tolerances at all the crucial internal interfaces, the engineer detects incipient errors immediately, and never needs to test the assembly as a whole to destruction. By continuously striving to improve the set points and tighten the tolerances at each internal interface, the quality of the whole product can be gradually improved. That is the essence of the six sigma quality improvement philosophy, which has been widely applied in manufacturing industry to increase product reliability at the same time as company profits.

set points and

and external

In the engineering of software, assertions at the interfaces between modules of the program play the same role as test probes in engine design. The analogy suggests that programmers should increase in the number and strength of assertions in their code. This will make their system more likely to fail under test; but the reward is that it is subsequently much less likely to fail in the field.

3

Macros

```
#ifdef DEBUG
#define ASSERT(b,str) {
  if (b) { }
  else {report (str);
    assert (false)}    }
#else #define ASSERT(b,str)
#endif
```

The defining characteristic of an engineering test probe is that it is removed from the engine before manufacture and delivery to the customer. In computer programs, this effect is achieved by means of conditionally defined macros. The macro is resolved at compile time in one of two ways, depending on a compile-time switch called DEBUG, set for a debugging run, and unset when compiling retail code. An assertion may be placed anywhere in the middle of executable code by means of this ASSERT macro. It is based on one that has been supplied to developers in Microsoft by my colleague Jon Pincus, in association with his program analysis tool PREfast. Many other examples in this talk are derived from the same collection of macros

on this slide

On retail code, the assertion is removed.
For a debugging run, the assertion is tested.
If true, it has no effect. If false, it calls
provides information for the debugging tools,

2

Explanations

- ASSERT(assertion, "reason why I think the assertion is true")
- Otherwise it's easy to forget.
- Helps both writer and reader.
- Pinpoints risk of similar errors
- Helps to avoid them in future

The whole point of an assertion is that the programmer should have a good reason for believing that it will always be true when the program is executed. When the programmer's reasoning has been confirmed by extensive experiments, we have a genuine scientific basis for confidence that the program is in fact correct. The programmer should be willing to explain the reason why the assertion is valid, as a second argument to the ASSERT macro. In case of error, discovered perhaps much later, the programmer will have the information needed to trace the underlying cause of the mistake. Correction may well be easier: but more than that. The programmer will be warned of other likely occurrences of a similar error in the existing code; and will be encouraged to improve the rigour of the reasoning, to avoid all such errors in the future.

That is why the PREfix assertion macro requires a second parameter, a string in which the programmer can explain quite informally the reason why the first parameter will always be true. The more obscure the reason, the greater the value of the explanation.

Documentation

- Protection for system against future changes

```
if (a >= b){ .. a++ ; .. };
    .. ..
  ASSERT(a != b, 'a has just
  been  incremented to avoid
  equality') ;
  x = c/(a - b)
```

A major activity of a software Company like Microsoft is the continuous evolution and improvement of old code to meet new market needs. Even quite trivial assertions, like that shown on this slide, give added value when changing the code. One Microsoft Development Manager recommends that for every bug corrected in test, an assertion should be added to the code which will fire if that bug ever occurs again. Some developers are willing to spend a whole day to design precautions that will avoid a week's work tracing an error that may be introduced later by a less experienced programmer. For example, the error message delivered on assertion violation in later evolution of the code can be carefully crafted to explain to later maintainers how the violation should have been avoided. Success in such documentation by assertions depends on long experience and careful judgment in predicting the most likely errors a year or more from now. Not everyone can spare the time to do this under pressure of tight delivery schedules. But it is likely that a liberal sprinkling of assertions in the code increases the accumulated value of commercial legacy code, when the time come to develop a new release.

for a subsequent release

Assumptions

- Used only during early test

```
SIMPLIFYING_ASSUMPTION
(strlen(input) < MAX_PATH,
  ''not yet checking for
  overflow'')
```

- Failure indicates test was irrelevant
- Prohibited in ship code

In the early testing of a prototype program, the developer wants to check out the main paths in the code before dealing with all the exceptional conditions that may occur in practice. In order to document such a development plan, PREfix provides a variety of assertion which is called a simplifying assumption. The quoted assumption documents exactly the cases which the developer is not yet ready to treat, and it also serves as a reminder of what remains to do later. Violation of such assumptions in test will simply cause a test case to be ignored, and should not be treated as an error. But the priority of the test case should be increased, to ensure that the eventual special case code will be adequately tested. Of course, in a retail build when the debug flag is not set, the macro will give rise to a compile-time error; it will not just be ignored like an ordinary assertion. This gives a guarantee against the risk incurred by more informal TO DO comments, which occasionally and embarrassingly find their way into ship code.

Optimisation

```
switch (condition) {
  case 0:  .. ..   ; break;
  case 1:   .. ..   ;break;
  default: UNREACHABLE('condition
    is really a boolean');}
```

- Compiler emits less code

Assertions can help a compiler produce better code. For example, in a C-style case statement, a default clause that cannot be reached should marked with an UNREACHABLE assertion, and the compiler avoids emission of unnecessary code for this case. In future, perhaps assertions will give further help in optimisation, for example by asserting that pointers or references do not point to the same location. Of course, if such an assertion were false, the effect could be awful; but fortunately it can be diagnosed quickly if the fault is reproduced on a debugging run, ~~because the assertion will detect it immediately.~~ And whenever the code is changed, it is subjected again to a massive suite of regression tests. For this reason, assertions are widely believed to be the only believable form of program documentation. I still hope that one day it will be possible to confirm such belief by proof.

two distinct

by falsity of the assertion

What is more, the assertion is likely to be kept up to date with the code

Assertions in retail code

- VSASSERT assertions are ignored
- VsVerifyThrow … generate exception
- VsVerify …user chooses

The original purpose of assertions was to ensure that program defects are detected as early as possible in test, rather than after delivery. But the power of the customer's processor is constantly increasing, and the frequency of delivery of software upgrades is also increasing. It is therefore more and more cost-effective to leave a certain proportion of the assertions in ship code; when they fire they generate an exception, and the choice is offered to the customer of sending a bug report to Microsoft. This is much better than a crash, which is a likely result of entry into a region of code that has never been encountered in test. A common idiom is to give the programmer control over such a range of options by means of different ASSERT macros. These three examples are taken from the Visual Studio project.

PROGRAM ANALYSIS

PREFIX_ASSUME

- Reduces PREFIX noise
- pointer = find (something);
  PREFIX_ASSUME (pointer != NULL,
  "see the insertion three lines back");
  … pointer ->mumble = blat …

Many programmers take advantage of high-level compiler warnings to remove program errors from their code.

in Microsoft The global program analysis tool called PREfix is now widely used by Microsoft development teams to detect program defects at an early stage, merely by scanning the program text, and even before compiling and testing. Typical defects detected by PREfix are a NULL pointer reference, an array subscript out of bound, a variable not initialised. PREfix ~~works by analysing~~ all paths through each method body, and it gives a report for each path on which there may be a defect. The trouble is that most of the paths considered can never in fact be activated. The resulting false positive messages still require considerable effort to analyse and reject; and the rejection is prone to error too.

It works. Others use analyses

special program analysis tools like RCLint.

Assertions can help the PREfix anomaly checker to avoid unnecessary noise. If something has only just three lines ago been inserted in a table, it is annoying to be told that it might not be there. The ASSUME macro allows the programmer to tell PREfix information about the program that cannot be automatically deduced.

23

Defect tracking

- Office Watson keys defects by assertions
- Integrates with RAID data base
- Identifies bugs across builds/releases
- Integral to the programming process

Assertions feature strongly in the code for Microsoft Office – around a quarter of a million of them. They are automatically given unique tags, so that they can be tracked in successive tests, builds and releases of the product, even though their line-number changes with the program code. Assertion violations are recorded in RAID, the standard data base of unresolved issues. When the same fault is detected by two different test cases, it is twice as easy to diagnose, and twice as valuable to correct. This kind of fault classification defines an important part of the team's programming process. [Kirk Glerum]

insertions and removals

24

Assertion Languages

- Bertrand Meyer
  - Eiffel, assertions as contracts
- Leavens, Baker, Ruby
  - Java Modelling Language
- Leino, Nelson, Saxe
  - ESC/Java, Extended Static Checker

I will now broaden my view away from current practice in Microsoft and towards the future progress of research into the use of assertions for the specification and verification of interfaces within a large software system; and in particular, the interface between class libraries and their users. In the design of Bertrand Meyer's Eiffel programming language, interface specifications are recommended as a sort of contract between implementers and users, in which each side undertakes certain obligations in return for corresponding guarantees. The same ideas are incorporated in draft proposals for assertion conventions recommended for specifying Java programs. Two examples are the Java modelling language and the Extended Static Checker. ESC is already an educational prototype of a verifying compiler.

designed

25

Interface assertions

- Used at least twice
- And again on each release
- Permits unit test of each module
- Permits modular analysis and proof

The references given on the previous slide give especial prominence to assertions at the major interfaces between modules of code. Assertions at interfaces give exceptionally good value. Firstly, they are exploited at least twice, by the implementer of the interface and by all its users. Secondly, interfaces are usually more stable than code, so the assertions that define an interface are used repeatedly whenever code is enhanced for a later release. Interface assertions permit unit testing of each module separately from its use; and they give good guidance in the design of rigorous test cases. Finally, they enable the analysis and proof of a large system to be split into smaller parts, separately for each module. This is absolutely critical. Even with fully modular checking, the first application of PREfix to a twenty million line product took three weeks of machine time; and even after a series of optimisations and compromises, it still takes three days.

Preconditions

```
void insert(node *n){
  PRECONDITION ( n != NULL &&
  invariant(), 'don't insert a
  non-existent object');
  SIMPLIFYING-ASSUMPTION
                  (find(n)== 0);
    .....
```

A precondition is defined as an assertion made at the beginning of a method body. It is the caller of the method rather than the implementer who is responsible for the validity of the precondition on entry; the implementer of the body of the method can just take it as an assumption. Recognition of this division of responsibility protects the virtuous writer of a precondition from having to inspect faults which have been caused by a careless caller of the method. As an example, consider the insertion of a node in a circular list, which may require that the parameter is not NULL. The example displayed above includes also a simplifying assumption; the assumption uses the find method local to the same class to check that the inserted object is not already there.

Post-conditions

```
   .. ..
POST_CONDITION ( find(n)&&
  invariant(), 'the inserted
object will be found in the
list')
 }
```

- obligation on method writer to verify

A post-condition is an assertion which describes (at least partially) the purpose of a method call. The caller of a method is allowed to assume its validity. The obligation is on the writer of the method to ensure that the post-condition is always satisfied. Preconditions and post-conditions document the contract between the implementer and the user of the class. This aspect of assertions has been heavily exploited in the Eiffel programming language.

on return from the call

Invariants

- True of every object …
- …before and after every method call
- `bool invariant ( )`
  {…tests that list is circular…}

The most useful kind of assertion is called

Assertions are particularly valuable for documenting object-oriented programs. An invariant ~~is defined as an assertion that~~ is intended to be true of every object of a class before and after every method call. It can be coded as a suitably named boolean method of the same class. For example, in a class that maintains a private list of objects, the invariant could state the implementer's intention that the list should always be circular. While the program is under test, the invariant can be retested after each method call, or even before as well, as shown on the previous two slides.

because

1

30

Invariants

- Integrity checking
- Software audits
- Post-mortem dump-cracking.

Invariants are widely used today in software engineering practice, though not under the same name. For example, every time a PC is switched on, or a new application is launched, invariants are used to check the integrity of the current environment and of the data held in long-term storage. In Microsoft Office, invariants on the structure of the heap are used to help diagnose storage leaks. In the telephone industry, ~~they~~ are tested in real time by a software auditing process, which runs concurrently with the switching software in an electronic exchange. Any call records that are found to violate the invariant are just re-initialised or even just deleted. It is rumoured that this technique once raised the reliability of a system from undeliverable to irreproachable.

invariants

In Microsoft, I see a future role for invariants in post-mortem dump-cracking, to check whether a failure was caused perhaps by some incident long ago that corrupted object data on the heap. Such a test has to made on the customer machine, because the heap is too voluminous to communicate the whole of it to a central server.

data base

Assertion inference: DAIKON

- Dynamic discovery of likely assertions by inference from data collected in test
- Gives warning of anomalies
- Estimates test coverage
- Helps when code is changed
- Michael Ernst

I have given you ~~some~~ a number of examples ~~of~~ illustrating the role of assertions in current programming practice. I hope that they will continue to give even greater benefits in future. But in order to exploit these advantages in the great mass of legacy code, it will be necessary to annotate it with more assertions than are commonly found at present. For this, some kind of mechanical assistance is practically essential.

In a recent Washington thesis submission, Michael Ernst describes how to generate assertions by machine inference from data collected in test. The programmer selects where to place the assertion, and which variables it should mention. The resulting assertion is reported back to the programmer. An anomaly is suspected if the inferred assertion is found to be unexpectedly weak. Alternatively, if the assertion states that a variable only ever takes one or two values, this may be stronger than intended; and it indicates that the test coverage should be expanded. Or maybe the code can be simplified to remove treatment of cases that in practice cannot occur. An inferred assertion can never by itself reveal an existing program defect; but it may do so when the code is changed for the next release.

35

Capabilities

- Declare cap_set as an abstract variable holding the set of permitted actions.
- Every action is preceded by an assertion that it is in the cap_set of the current thread.
- Some actions increase or reduce cap_set.
- Tools are available to reliably insert these assertions and actions.

Assertions of the traditional kind are poor at recording temporal obligations, like 'you must open a file before reading it'. The solution to this problem is to introduce extra assertional variables and assignments into a test harness. They are sometimes called model variables or ghost variables or history variables. Permissions and obligations are made explicit in an abstract assertional variable, let's call it cap_set, containing the set of actions permitted to the current thread. Every action must be preceded by an assertion that this action is in the cap_set. Some actions like opening a file are defined to increase the capability set, and others to reduce it. On other occasions, a capability is passed from one method or thread to another. A program that satisfies all the additional assertions about capabilities is still likely to be correct, even when the cap-set variable is sliced out of ship code. Tools will be needed for automatically inserting the extra variables, assignments and assertions.

Test case generation

- UTAHlite (Jason Taylor)
- Model-based testing
- Desired behavior abstracted as a graph
- With actions on the edges
- Generates test scripts
- Drives automated test suites
- Checks concurrent interactions

The generation of really difficult test cases is essential to ~~the use of~~ testing to establish confidence in the correctness of programs. The aim of a test is always to satisfy all the preconditions and simplifying assumptions of a program, but to violate one of the post-conditions. The automatic generation of such tests is the ultimate goal of research in testing tools. In general, this is just as difficult as proving the correctness of the assertion; in fact, it would be a most desirable side-effect of a failed attempt at proof.

But a start has already been made in the development of an Universal Test Automation Harness UTAH. This is used to specify the behaviour of the user of the unit under test in the form of a graph, with nodes representing the states of the system/user, and arrows labeled by the actions that take the system from one state to the next. The graph is then interrogated by the test engineer to generate test scripts exhausting a wide range of possibilities.

This kind of Finite State Machine model can be used to define protocols at internal interfaces of a large system, perhaps even between concurrent programs. The state of the machine is held as the value of a ghost variable, updated by each relevant event in the life of the program. The state machine model can then be used either to prove correctness, or at least to generate assertions that detect failure, and indicate which side of the interface is responsible.

40

Model checking

- Automatically Validating Temporal Safety Properties of Interfaces. (SLAM)
- Thomas Ball and Sriram K. Rajamani
- Uses symbolic execution to generate the necessary assertions
- Proves them by model checking
- Or generates a failing test case

Model checking also has an important role in increasing confidence in the correctness of programs. One example is the SLAM project, now progressing in Microsoft Research. This is initially targeted at the security of device routines, written by independent hardware vendors and submitted for certification by Microsoft. The automatic program starts by inserting test probes based on the capability sets that I have described ~~just now~~. It then examines all execution paths that would violate one of the security constraints. It automatically generates any assertions that would be necessary to exclude that violation. It then calls on a modern high-speed model-checker to see if it can prove the validity of these assertions. If not, a stronger set of assertions has to be generated, until success is achieved. Or if failure is reported, a test case should be generated that would reveal it. As I mentioned before, that is very difficult. As an alternative, it is always possible to leave the generated assertions in the code, at least during test, to trigger an error report rather than a crash.

already

2

42

Program verification

- Extended Static Checking (of Java)
- Greg Nelson and Rustan Leino
- Generates verification conditions
- Proves them by decision procedures
- Validates omission of assertions

The ultimate goal of programmer productivity tools is to reduce the costs and delays associated with programming error, and raise product quality by delivering code that is almost free of defects. Mathematical proof of program correctness would be the ultimate achievement, and like all extremes, it serves as the inspiration and goal and criterion of long-term academic research.

Greg Nelson has recently released for academic use a checker that will in many cases prove correctness of assertions, thereby justifying their omission even in debug builds (but not in the source text!). Improvement in the power of the built-in decision procedures may eventually extend this technique to an industrial scale. In fact, I see the possibility of a continuous evolution of the mathematical sophistication of program analysis tools, until they come to play a role close to that of Floyd's original verifying compiler.

like PREfix

Conclusion

- Science is concerned with general theories,
- It seeks ideals of truth and certainty, based on mathematical deduction and proof.

- Engineering is concerned delivery of a particular product, at a given time and cost.
- It requires common sense, judgement, and compromise.

In conclusion, I would like to relate my message to the theme of this conference. I have devoted most of my research career to the hard disciplines of the mathematical sciences. That was my choice, and I do not need to justify it. But in this talk I have tried to complement the rigours of fundamental research by describing some of the ordinary ways in which the discoveries of that research are exploited by ordinary engineers. ~~The~~ mathematical rigours need to be softened by considerations of common sense, by engineering judgement, by use of computerised design tools, and even by managerial or political compromise. The challenges of exploitation are just as hard as those of science; they are met by appealing to particular circumstances of each case; and lapses in rigour are excused by considerations of cost and time-scale, which have no place in the pursuit of pure science.

Acknowledgements

Rick Andrews, Chris Antos, Tom Ball, Pete Collins, Terry Crowley, Mike Daly, Robert Deline, John Douceur, Sean Edmison, Kirk Glerum, David Greenspoon, Yuri Gurevich, Martyn Lovell, Bertrand Meyer, Jon Pincus, Harry Robinson, Hannes Ruescher, Marc Shapiro, Kevin Schofield, Wolfram Schulte, David Schwartz, Amitabh Srivastava, David Stutz, James Tierney, Jason Zions

My thanks to all my new colleagues in Microsoft Research and Development who have told me about their current use of assertions in programming and testing. Acknowledgments also to all my colleagues in academic research, who have explored with me the theory of programming and the practice of software engineering. In my present role as Senior Researcher in Microsoft Research, I have an extraordinary privilege of witnessing and maybe even slightly contributing to the convergence of these two developments, and seeing results that contribute back to both theory and to the practice of programming.