Towards the Verifying Compiler

Tony Hoare
Microsoft Research Ltd., Cambridge, UK. March 2002

Summary.

A verifying compiler is one that proves automatically that a program is correct before allowing it to be run. Correctness of a program is defined by placing assertions at strategic points in the program text, particularly at the interfaces between its components. From recent enquiries among software developers at Microsoft, I have discovered that assertions are widely used in program development practice. Their main role is as test oracles, to detect programming errors as close as possible to their place of occurrence. Further progress in reliable software engineering is supported by programmer productivity tools that exploit assertions of various kinds in various ways at all stages in program development. The construction and exploitation of a fully verifying compiler remains as a long-term challenge for twenty-first century Computing Science. The results of this research will be of intermediate benefit long before the eventual ideal is reached.

Introduction.

An understanding of the role of assertions in Checking a Large Routine goes back to Alan Turing in 1950. The idea of a verifying compiler, which uses automatic theorem proving to guarantee the correctness of the assertions, goes back to a 1967 article by Bob Floyd on Assigning Meanings to Programs. And the idea of writing the assertions even before writing the program was propounded in 1968 by Edsger Dijkstra in an article on a Constructive Approach to the Problem of Program Correctness. Dijkstra's insight has been the inspiration of much of the research in formal methods of software engineering conducted in University Computing Departments over the last thirty years.

An assertion in its most familiar form is a Boolean expression that is written as an executable statement 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 by definition incorrect. But if all assertions always evaluate to true, then at least no program defect has ever 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 certainly correct, at least insofar as correctness has been captured by the assertions embedded in it. The construction and validation of such proofs are the goal of the verifying compiler.

Early attempts to implement a verifying compiler were frustrated by the inherent difficulties of mechanical theorem proving. These difficulties have inspired productive research on a number of projects, and with the aid of massive increases in computer power and capacity considerable progress has been made. I suggest that an

intensification of co-operative research efforts will result in the emergence of a workable verifying compiler some time in the current century.

A second problem is that meaningful assertions are notoriously difficult to write. This means that work on a verifying compiler must be matched by a systematic attempt to attach assertions to the great body of existing software libraries and services. The ultimate goal is that all the major interfaces in freely available software should be fully documented by assertions approaching in expressive power to a full specification of its intended functionality. Such a major long-term challenge accords well with the ideals of the proponents of open software, whose quality is assured by contributions from many sources.

But would the results of these related research projects ever be exploited in the production of software products used throughout the world? That depends on the answer to three further questions. Firstly, will programmers use assertions? The answer is yes, but they will use them for many other purposes besides verification; in particular, they are already widely used to detect errors in program test, as I shall describe in the next section. The second question is: will they ever be used for verification of program correctness? Here, the answer is conditional: it depends on wider and deeper education in the principles and practice of programming, and on integration of verification and analysis tools in the standard software development process. And finally, the sixty-four billion dollar question: is there a market demand for the extra assurances of reliability that can be offered by a verifying compiler?

For many years, my friends in the software industry have told me that in all surveys of customer requirements the top two priorities have always been firstly an increase in features, and secondly an increase in performance. Reliability takes only the third place. But now it seems that widely available software already provides enough features to satisfy nearly all demands, and widely available hardware already satisfies most demands for performance and capacity. The main remaining obstacle to the integration of computers into the life of society is a wide-spread and well-justified reluctance to trust the software. A recent email by Bill Gates to Microsoft and all its subsidiaries has put trustworthy computing at the head of the agenda. This policy statement has already been put into force by devoting the efforts of the entire Microsoft Windows team during the whole month of February to a security drive. Expensive it is, but not as expensive as some recent viruses like Code Red, which have led to world-wide losses estimated at over a billion dollars.

Assertions in Program Testing.

In my thirty years as an academic scientist, I pursued the traditional academic ideals of rigour and precision in scientific research. I sought to enlarge our understanding of the theory of programming to show that large-scale programs can be fully specified and proved to be correct with absolute mathematical certainty. I hoped that increased rigour in top-down program development would significantly reduce if not eliminate the burden of program testing and debugging. I would quote with approval the famous dictum of Dijkstra, that program testing can prove the existence of program bugs, but never their absence.

A very similar remark was made by the famous philosopher Karl Popper. His Philosophy of Science is based on the principle of falsification, namely that a scientific theory can never be verified by scientific experiment; it can only be falsified. I accept his view that a scientific advance starts with a theory that has some a priori grounds for credibility, for example, by deduction from the principle 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 with high confidence. A well tested theory will be used with confidence to help in the formulation and test of further and more advanced theories, and in the design of experiments to refute them.

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 perhaps, 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 staff on each project.

This account of the vital role of testing in the progress of science is reinforced by consideration of the role of test in engineering. In all branches of engineering, rigorous product test is an essential prerequisite before shipping a new or improved product to the customer. 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 and external 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 interface, the quality of the whole product can be gradually raised. That is the essence of the six sigma quality improvement philosophy, which has been widely applied in manufacturing industry to increase profits at the same time as customer satisfaction.

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.

In the three years since I retired from academic life, I have been working in the software industry. This has enabled me to balance the idealism that inspires academic research with the practical compromises that are essential to industrial engineering. In particular, I have radically changed my attitude towards program testing which I now understand to be entirely complementary to scientific design and verification

methods, and makes an equal contribution to the development of reliable software on an industrial scale. It is no accident that program testing exploits the same kind of specifications by assertions that form the basis of program verification.

Assertions in current Microsoft Development Practice.

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 a conditionally defined macro. 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 that will be shipped to the retail customer. An assertion may be placed anywhere in the middle of executable code by means of this ASSERT macro.

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

In addition to their role in product test, assertions are widely recommended as a form of program documentation. This is of vital concern to major software suppliers today, because their main development activity is the continuous evolution and improvement of old code to meet new market needs. Even quite trivial assertions, like the following, give added value when changing the code.

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

One Development Manager in Microsoft 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. Ideally, there should be enough assertions in a program that nearly all bugs are caught by assertion failure, because that is much easier to diagnose than other forms of failure, for example, a crash. 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, when the code is modified by a less experienced programmer. 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 would increase the accumulated value of legacy, when the time comes to develop a new release of the software.

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, some developers have introduced a variety of assertion which is called a simplifying assumption.

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

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. Of course, in compiling retail code for delivery to the customer, the debug flag is not set; and then 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 comments and messages about known bugs and work that is still TO DO; such comments occasionally and embarrassingly find their way into code shipped by Microsoft.

All the best fault diagnoses are those given at compile time, since that is much cheaper than diagnosis of errors by test. In one product team in Microsoft, a special class of assertion has been implemented called a compile-time check, because its value, true or false, can be computed at compile time.

```
COMPILE_TIME_CHECK (sizeof(x)==sizeof(y), 'addition is undefined for arrays of different sizes')
```

The compile time error message is generated by a macro that compiles to an invalid declaration (negative array bound) in C; of course, the assertion must be restricted to use only values and functions computable by the compiler. (The compiler will still complain if not). The example above shows a test of conformity of the size of two array parameters for a method. Of course, as we make progress towards a verifying compiler, the aim will be to increase the proportion of assertions whose violation will be detected at compile time.

Assertions can help a compiler produce better code. For example, in a C-style case statement, a default clause that cannot be reached can be marked with an UNREACHABLE assertion, and a compiler (for example Visual C) 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 of the optimisation could be awful. But fortunately assertions which have been frequently tested are remarkably reliable; indeed, they are widely believed to be the only believable form of program documentation. When assertions are automatically proved by a verifying compiler, they will be even more believable.

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

A global program analysis tool called PREfix is now widely used by Microsoft development teams. Like PCLint and Purify, its role is to detect program defects at the earliest possible stage, even before the program is compiled. Typical defects 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 can never in fact be activated. The resulting false positive messages are called noise, and they still require considerable human effort to analyse and reject; and the rejection of noise is itself highly prone to error. It is rumoured that the recent Code Red virus gained access through a loophole that had been detected by PREfix and deliberately ignored.

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. A special ASSUME macro allows the programmer to tell PREfix relevant information about the program that cannot at present be automatically deduced.

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

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.

In Microsoft, over half the effort devoted to program development is attributed to test. For legacy code, there is an accumulation of regression tests that are run for many weeks before each new release. It is therefore very important to select tests that are exceptionally rigorous, so as to increase the chance of catching bugs before delivery. Obviously, tests that have in the past violated assertions are the most important to run again. Violation of a simplifying assumption is a particular reason for increasing the priority of a test, because it is likely to exercise a rare and difficult case.

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 shipped code; when they fire they generate an exception, and the choice is offered to the customer of sending a bug report to Microsoft. The report includes a dump of the stack of the running program. About a million such reports arrive in Redmond every day for diagnosis and correction. A controlled dump 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.

Assertions in programming languages.

The examples of the previous section have all been implemented as macro definitions by various teams in Microsoft, and each of them is used only by the team which implemented them. In the code for Microsoft Windows, we have found over a thousand different assertion macros. This is a serious impediment to the deployment of programming analysis tools to exploit assertions. The best way of solving this problem in the long term is to include an adequate range of assertions into the basic programming language. A standard notation is likely to be more widely accepted, more widely taught, and more widely used than a macro devised by an individual programmer or programming team. Furthermore, as I suggested when I first started research on assertions, provision of support for sound reasoning about program correctness is a suitable objective criterion for judging the quality of a programming language design.

Significant progress towards this goal has been made by Bertrand Meyer in his design of the Eiffel programming language. Assertions are recommended as a sort of contract between the implementers and the users of a library of classes; each side undertakes certain obligations in return for corresponding guarantees from the other. The same ideas are incorporated in draft proposals for assertion conventions adapted for specifying Java programs. Two examples are the Java modelling language JML (Leavens, Baker and Ruby) and the Extended Static Checker ESC for Modula 3 and Java (Nelson, Leino and Saxe). ESC is already an educational prototype of a verifying compiler.

Assertions at interfaces presented by a library give exceptionally good value. Firstly, they are exploited at least twice, once by the implementer of the interface and possibly many times by all its users. Secondly, interfaces are usually more stable than code, so the assertions that define an interface are used repeatedly whenever library code is enhanced for a later release. Interface assertions permit unit testing of each module separately from the programs that use it; and they give 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, devised and checked 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 to complete the analysis; and even after a series of optimisations and compromises, it still takes three days.

Three useful kinds of assertions at interfaces are preconditions, postconditions and invariants. 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. In the design of test cases for unit test, each case must be generated or designed to satisfy the precondition, preferably at the edges of its range of validity.

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. Test cases for unit test must be generated or designed with the best possible chance of falsifying the postcondition. In fact, postconditions and other assertions should be so strong that they are almost certain to find any defect in the program. As with a scientific theory, it should be almost inconceivable that an incorrect program will escape detection by one of the tests.

In object oriented programs, preconditions and post-conditions document the contract between the implementer and the user of the methods of a class. The interface between successive calls of different methods of an object of the the class is specified by means of an invariant. An invariant is defined as an assertion that is intended to be true of every object of a class at all times except while the code of the class is executing. It can be specified as a suitably named boolean method of the same class. An invariant does not usually feature as part of the external specification of a class; but rather describes the strategy of the implementation of the individual methods. 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.

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 stored data base. In the Microsoft Office project, invariants on the structure of the heap are used to help diagnose storage leaks. In the telephone industry, they are used 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 deleted. It is rumoured that this technique once raised the reliability of a newly developed system from undeliverable to irreproachable.

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 data on the heap. This test has to be made on the customer machine, because the heap is too voluminous to communicate the whole of it to a central server. There is a prospect that the code to conduct the tests will be injected into the customer's software as the occasion demands. The scale of the current problem of dumps is easy to calculate. With the target Microsoft customer base, it is not long before the number of dumps arising from anomalies at the customer site could exceed one million dumps per day.

Summary and conclusion.

The primary use of assertions today is for program instrumentation; they are inserted as probes in program testing, and they serve as a test oracle to give early warning of program defects, close to the place that they occur. They are also used for program documentation, to assist later developers to evolve the product to meet new market needs. In particular, they specify interfaces between major software components, such as libraries and application programs. Assertions are just beginning to be used

by the C compiler in code optimisation. They are used to classify and track defects between customer sites, between test cases, and between code changes. Assertions are being introduced into program analysis tools like PREfix, to raise the precision of analysis and reduce the noise of false positives. Increasingly, assertions are shipped to the customer to make a program more rugged, by forestalling errors that might otherwise lead to a crash.

At present, Microsoft programmers have achieved each of these benefits separately. This gives rise to an attractive opportunity to all the benefits together, by reusing the same assertion again and again for different purposes, one after the other. In this way, they will be encouraged to introduce assertions as early as possible into the development process. They can then play a guiding role in a top-down process of program design, as suggested in Dijkstra's original constructive approach to correctness.

I expect that assertions will bring even greater benefits in the future, when they are fully supported by a range of programmer productivity tools. They will help in deep diagnosis of post-mortem dumps. They will serve as a guide in test case generation and prioritisation. They will help to make code concurrency-safe, and to reduce security loop-holes. In dealing with concurrency and security, there is still plenty of scope for fundamental research in the theory of programming.

In conclusion, I would like to re-iterate the research goal which I put forward when (in 1968) I first embarked on research into program correctness based on assertions. It was to enable future programming languages and features to be designed from the beginning to support reasoning about the correctness of programs, in the hope that this would provide an objective criterion for evaluating the quality of the design. I believe that modern language designers, including the designers of Java and C#, are beginning to recognise this as a valuable goal, though they have not yet had the brilliant idea of using assertions to help them achieve it. As a result, these languages still include a plethora of fashionable features, and low-level constructions which are supposed to contribute to efficiency. These make it difficult or impossible to use local reasoning about the correctness of a component, in the assurance that correctness will be preserved when the components are assembled into a large system.

Fortunately, these problems are soluble even without a switch to a more disciplined programming language. Program analysis tools like PREfix show the way. By conducting an analysis of the source code for the entire system, it is possible to identify the use of the more dangerous features of a programming language, identified as those which violate modularity and invalidate normal correctness reasoning. A notorious example is the introduction of aliasing by passing the same (or overlapping) parameter more than once by reference to the same procedure call. Such violations are flagged by a warning message. Of course, the warnings can be ignored. But in Microsoft, at least, there is a growing reluctance to ignore warning messages. It is a brave programmer who has the confidence to guarantee program correctness in the face of such a warning, when the penalty for incorrectness is the introduction of a virus that causes a billion dollars of damage. And when all programmers rewrite their code to eliminate all such warnings, they are effectively already using a much improved programming language.

These are the reasons for optimism that professional programmers in the software industry will be ready to accept and use a verifying compiler, when it becomes available. In industry, work towards the evolution of a verifying compiler will progress gradually by increasing the sophistication of program analysis tools. But there is a splendid opportunity for academic research to lead the way towards the longer term future. I have already mentioned the verifying compiler as one of the major challenges of Computing Science in the twenty first century. To meet the challenge we will need to draw on contributions from all the different technologies relevant to mechanised proof. Like the Human Genome project, or the launch of a new scientific satellite, or the design of a sub-atomic particle accelerator, progress on such a vast project will depend on a degree of collaboration among scientists that is so far unprecedented in Computer Science.

There is now a great mass of legacy software available as test material for evaluating progress. Work can start by annotating and improving the quality of the interfaces to the base class libraries, which come with the major object oriented languages. The work will be meticulous, exhausting and like most of scientific research, it will include a large element of routine. It will require deep commitment, and wide collaboration, and certainly the occasional breakthrough. Fortunately, the goal of the project is closely aligned with the ideals of the open source movement, which seeks to improve quality by contributions from many workers in the field.

We will also need to recognise the complementary role of rigorous program testing; we must integrate verification with all the other productivity tools that are aimed at facilitating the program development process, including the maintenance and enhancement of legacy code. I expect that the use of full program verification will always be expensive; and the experienced software engineer will always have to use good engineering judgement in selecting a combination of verification and validation techniques to achieve confidence in correctness, reliability and serviceability of software. For safety critical software, the case for full verification is very strong. For operating system kernels and security protocols, it is already known that there is no viable alternative. For assurance of the structural integrity of a large software system, proof of avoidance of overflows and interference is extremely valuable. There will also be many cases where even a partial verification will permit a significant reduction in the volume and the cost of testing, which at present accounts for more than half the cost of software development. Reduction in the high cost of testing and reduction in the interval to delivery of new releases will be major commercial incentives for the expansion of the role of verification; they will be just as persuasive as the pursuit of an ideal of absolute correctness, which has been the inspiration of scientific endeavour. In this respect, software engineering is no different from other branches of engineering, where well-judged compromises in the light of costs and timescales are just as important as an understanding of the relevant scientific principles.

The fact that formal software verification will not by itself solve all the problems of software reliability should not discourage the scientific community from taking up the challenge. Like other major scientific challenges, the appeal of the project must be actually increased by its inherent difficulty, as well as its contribution to the old academic ideals of purity and integrity, based on the enlargement of understanding and the discovery and exploitation of scientific truth.

Acknowledgements.

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. Their names include 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.

Acknowledgments also to all my colleagues in Oxford and many other Universities, 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 the extraordinary privilege of witnessing and maybe even slightly contributing to the convergence of academic and industrial developments, and I have good hope of seeing results that contribute back to the development of the theory and also to the practice of programming.