Tools and technology for trusted code.

Jon Pincus and Tony Hoare. February 2002.

Programmer productivity tools like PREfix and PREfast are already making a valuable contribution to the current Microsoft drive to justify and increase our customers' trust in our code. This note describes the technology appropriate to intensifying this drive, in a way that is ~~also conducive~~ to programmer productivity.

The key concept is the instrumentation of interfaces between the modules, assemblies and components of a large system by means of probes, which continuously monitor the safety, the behaviour and the consistency of each part during a run of the system as a whole. The programs which define the probes are at present run ~~on the same machine as~~ the system itself; and it is too expensive to run all of them all the time. Means must be provided to switch them off when sufficient trust has developed in the assembly on one or both sides of the interface – and switch them on again when a change in the code (or worse, a detected error) diminishes that trust. Our ultimate goal is that trust in the whole system can be confirmed by program verification technology, which guarantees which of the probes can be safely turned off in ship code. In this way, the probes will ~~play the same~~ role as the compile-time type-checking in a modern programming language, ~~which~~ makes the call of methods far safer than in older languages like FORTRAN and C.

Test probes are already widely used by Microsoft developers in the guise of assertion macros, which are conditionally compiled for test code, and ignored in ship code. A recent count has revealed over a million assertions in the code for Windows XP. Over

a declaration declarations

thousand different ~~conditionally compiled~~ assertion macros have been written by different product teams. It is a goal of tools currently under development to exploit and extend the expertise of Microsoft Developers in the use of assertions, to generalize the concept of an assertion to cover behavioural ~~security~~ properties, security and to rationalize the notation for assertions across the entire code base, and to make these benefits available to (and use of) users of C#. and concurrency safety,

Assertions, the existing expertise.

The main use of assertions today is to assist in program test and fault diagnosis. In all branches of engineering, product test is an essential prerequisite before release to manufacture of 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 raised. That is the reason for the success of the six sigma quality improvement initiative, at least in Companies engaged in more traditional forms of engineering.

In the engineering of software, it is the assertions at the interfaces between modules of the program that play the same role as test probes in engine design. My analogy with tightening tolerances suggests that programmers who wish to write trustworthy programs should gradually increase the number and strength of the assertions in code. Paradoxically, the intended effect of assertions

that they put

is to make a system **more** likely to fail under test; but the reward is that failure is much **less** likely after shipping to the customer.

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.

In addition to their use as test probes, assertions play an important role in program documentation. This is particularly valuable for Microsoft, because our main business is now the continuous evolution and improvement of old code to meet new market needs. Even quite trivial assertions give added value when the time comes to change the code for the next release. One Development Manager (Hannes Ruescher) recommends that for every bug corrected in test, an assertion should be added to the code which will fire if that bug is ever reintroduced by future changes in the code. An even stronger recommendation: From the beginning, there should be enough assertions in the code to ensure that nearly all bugs will be caught by assertion failure – much easier to diagnose than any other failure.

Some developers (Chris Antos) are willing to spend a whole day to design precautions that will avoid a week's work by a less experienced programmer, tracing an error that may be introduced by a later change to the code. 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 the current liberal sprinkling of assertions in the code is a significant contribution to

the accumulated value of Microsoft legacy, making it more adaptable for a new release.

Here is an unexpected use for assertions, suggested by Marc Shapiro. 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, PREfast 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 protection against the risk incurred by more informal TO DO comments, which occasionally and embarrassingly find their way into retail code.

It is sometimes useful for an assertion to refer to the previous values of variables, or to a log of significant actions that a program has performed. That is the purpose of the assertional macro provided with PREfast; its parameter may consist of arbitrary declarations of variables and assignments to them. Typically, such a variable is used to hold the initial value of a parameter or object property, so as to check in the postcondition that its value has been correctly changed by the body of a method.

All the best debug messages are those given at compile time, since that avoids all the hassle of diagnosis of errors by test. In the Windows product team, a special class of assertion has been implemented called a compile-time check, because it can be evaluated at compile time. The compile time error message is

generated by a macro that compiles to an invalid declaration (negative array bound) in C in the case that the compiler evaluates the assertion to false; of course, the assertion must be one that uses only values and functions computable by the compiler. (The compiler will still complain if not.)

Of course, only a very few assertions can be evaluated at compile time – at present. To change this is exactly the long-term goal of future programmer productivity tools. By more sophisticated program analyses, it will be increasingly possible to guarantee (with mathematical certainty) that each assertion will be true on every occasion that it is evaluated. If this guarantee cannot be given, the tool should ideally generate a test case automatically that will expose the fault.

Of course, an assertion that has been proved to be always true can be optimised away, to avoid the overhead of evaluation, even on test runs. Because it is known that there will be no errors left for run-time testing. The state of the art in theorem proving technology is not yet sufficiently advanced to achieve this desirable goal, and may always be too expensive for universal application.

The global program analysis tool 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.

Assertions 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 information about the program that cannot be automatically

deduced. This is a much better way of reducing noise than just switching off the warning. This macro is not yet widely used.

Assertions can help a compiler produce better code. For example, in a C-style ~~case~~ switch statement, a default clause that cannot be reached can be 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. This will encourage the compiler to continue optimisation, in spite of the risk of an alias. The optimisation depends on confidence in the validity of the assertion. At present this confidence is built up on massive testing – in fact, assertions are widely believed to be the only reliable form of program documentation. When assertions are automatically proved by an analysis tool, they will be even more believable.

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 at all times, except while the code of the class is actually running. The present practice is to code an invariant 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.

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 dynamically allocate storage on 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 simply re-initialised or even just deleted. It is rumoured that this technique once raised the reliability of a system from undeliverable to irreproachable.

Tomorrow

We can 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 be made on the customer machine, because the heap is too voluminous to communicate the whole of it to a central server.

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. Office Watson automatically records and classifies assertion violations in RAID. 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.

Tomorrow

The original purpose of assertions was to ensure that program defects are detected as early as possible in test, rather than later-on, after check-in, after code complete, or even after delivery. But the power of the customer's processor is constantly increasing, and the frequency of delivery of software upgrades in the dot.NET environment is also increasing. It is therefore more and more cost-effective to leave a certain proportion of the assertions in retail 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 you already know~~ has never been encountered in test. A common idiom is to give the programmer control over

1

such a range of options by means of different ASSERT macros. In libraries provided by Microsoft to its customers, most of the preconditions will be SHIP-ASSERTS like this.

In future, we may expect that the decision whether to ship an assertion can be made later, even after the code is released. Object code manipulation tools like VULCAN will be able to inject assertions into code where necessary, even on the customer site.

Interfaces, the way of the future.

Assertions written at the interfaces between program modules, assemblies and components give exceptionally good value. Firstly, they are exploited at least twice, by the implementer of the interface and by its user – indeed by all its users. Secondly, interfaces are usually more stable over releases than the code, so the assertions that define an interface are used repeatedly whenever code is enhanced for a later release. This should make it safer for the users of a library to read the interface documentation than the code itself. 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, so that each part can be analysed separately in a modular fashion. This is absolutely critical. Even with fully modular checking, the first application of PREfix to Windows 2000 took three weeks of machine time; and even after a series of optimisations and compromises, it still takes three days.

The first important kind of assertion that one sees at an interface is a precondition. 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 every entry to the method; the implementer of

the body of the method can just take it as an assumption. Recognition of this division of responsibility across the interface protects the virtuous writer of a method from being called out 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 shown on this slide includes also a test of the class invariant and a simplifying assumption; the assumption uses the find method local to the same class to check that the inserted object is not already there.

The second main kind of interface assertion is the postcondition, defined as an assertion evaluated on return from the method. ~~The postcondition is an assertion which~~ It describes (at least partially) the purpose of a method call. The caller of a method is allowed to assume its validity on return from the call. The obligation is on the writer of the method to ensure that the post-condition is always satisfied, and that the class invariant is satisfied too. Preconditions and post-conditions document the contract between the implementer and the user of the methods of a class. This aspect of assertions has been heavily exploited in the Eiffel programming language.

In future, defect tracking in the style described above for Office will be assisted by the distinction between preconditions and postconditions. Violation of a precondition will be attributed to the calling program, whereas violation of a postcondition or invariant will be attributed to the called method. At present, Office Watson has no way of making this vital distinction.

Ordinary in-line assertions can also be regarded as interface assertions, lying on the boundary of the code that comes before them and the code that comes after. The distinctions between the different kinds of assertion are useful when it comes to the decision which assertions to switch off. In principle, the assertions

on an interface

should be tested for the less trusted side of each interface. When a method has just been written, all its internal assertions should be switched on. The invariant can be thought of as attached to the interface between all the public methods of a class. To begin with, it should be tested between every call of any method of the class. Preconditions and postconditions sit on the interface between a class and its user. If the user is less trusted than the class, the preconditions should be tested. If the class has been recently written or recently changed, it will certainly be less trusted than the regression tests that exercise it, and so the postconditions should also be tested. In future, we foresee tools that will assist in the selective allocation of trust levels to code under development, and the selective disablement of probes in conformance with project management policy.

Conventional assertions are restricted to testing the properties of a single machine state, the one in which it is evaluated. This is a serious restriction, and must be relaxed. We have already mentioned one example, where a postcondition needs to test the current value of a variable against a previous value. But this case needs to be generalised, because many important security properties of a program can be defined only in terms of a trace of its long-term behaviour. Any move towards concurrent programming will reinforce this necessity.

The technology that seems to be most effective is that of finite state machines, whose transitions are fired by interactions between the assemblies on each side of the interface. Examples are inputs and outputs, events and exceptions, method calls and returns. This technology has already been applied to the static checking of device drivers in the SLAM project. It is the basis of an automatic test case generator developed by Jason Taylor for use in IE. It is proposed to standardise on a notation and tool suite for the expression, display and analysis of finite state machines, equally useful for all these purposes.