Some Remarks about Random Testing

B A Wichmann,
National Physical Laboratory,
Teddington, Middlesex, TW11 0LW, UK
E-mail: baw@cise.npl.co.uk

May 1998

1 Introduction

The use of a pseudo-random number generator to produce test cases for software appears not to be widely used, although the method has been reported in the literature for many years [1, 2, 4]. This paper draws on experience with this technique in two areas: compilers and software components.

The use of a pseudo-random number generator to produce test cases which are expected to reflect actual usage of the software is not discussed here (we would call this statistical testing). Although such testing would have the advantage that reliability data could be produced, in practice with most software of any complexity, it is not possible to model accurately enough the distribution of real input data. Hence the test case generation considered here is for conventional testing purposes, rather than predicting reliability.

All the work here has exploited a portable and efficient random number generator [5]. Use of this generator has the advantage that one can reproduce test cases from the three integer seed values, which are input to the test case generator, no matter how complex the test cases are.

2 Compiler Stress Testing

Testing compilers is difficult due to the complexity of the processing they perform and also since it is virtually impossible to test components of a compiler in isolation.

One 'standard' method of compiler testing is to apply a fixed set of tests designed to show compliance with the international standard. Such validation suites are available for the standard languages such as Pascal [3] or Ada [7]. Both these validations suites are quite comprehensive, but it is nevertheless possible to have many simple errors in an immature compiler that passes these tests.

Using a test generator which produces self-checking, semantically correct programs is an attractive alternative to supplement the validation suites. In fact, although validation suites can (and do) check the front-end of compilers reasonably well, the back-end is much more difficult to test effectively.

One can gain some insight into back-end compiler bugs from a bug located in the Algol W compiler many years ago. Here, the registers were allocated in increasing order for the stack pointers, and decreasing order for temporaries in an expression. If

the registers were exhausted, the compiler issued an error message to state that the program was too complex. The error came when an odd-even pair of registers was required for integer divide *and* the number of registers left was very near to exhaustion. Incorrect code was produced, which remained undetected for ten years! This illustrates that back-end errors in a compiler can have a very complex relationship with the source text which makes in infeasible to test that back-end by conventional means via hand-written source text input. Equally, the errors cannot be found by testing the back-end in isolation, since the front-end is the only effective means of producing input for the back-end.

Bearing in mind the above problem, I wrote a Pascal Program Generator which *would* have found the Algol W type of error in a Pascal compiler. The idea behind this was to supplement the Validation Suite in the testing of Pascal compilers.

The only major problem with producing language generators is to ensure the program produced is semantically correct. For instance, one must ensure that expressions do not overflow. Having produced such a generator, stress testing a compiler is straightforward. A summary of the results from many tests run at NPL is in Table 1.

NPL has produced language generators for Pascal [9], Ada 83, Haskell, CHILL and a subset of Pascal. Three demanding sets of tests were applied to three different Ada compilers with amazingly good results — only minor bugs were found. In contrast, all the other generators when applied to relevant compilers found 'safety' bugs very quickly. By a safety bug, I mean generating incorrect object code from correct source code.

Developing the Ada generator was much more work than the others, since the static semantics of the language is larger, and the goal was to generate all the major parts of the language. Two problems have arisen with the generator: firstly, it has not yet been updated for Ada 95, and secondly, the generator very occasionally generates incorrect Ada (in fact, not observing restrictions on staticness which are different for Ada 83 and Ada 95). This later point is not a significant barrier to the use of the tool, since the incorrect Ada programs are rejected by the compiler (and even if they were not due to an error in the compiler, the run-time checks would still be 'correct').

All these test case generators contain facilities to adjust the length and complexity of the source code so that the compiler can be conveniently stress-tested. Some compilers limit the complexity of programs that can be compiled to quite modest limits which restricts the stress-testing that can be undertaken. The definition of stress-testing from the IEEE glossary reads: 'Testing conducted to evaluate a system or component at or beyond the limits of its specified requirements'. However, compilers typically do not specify the limits on the complexity of the programs they will handle and therefore the 'requirements' issue is unresolved.

Once such a generator is available, the testing soon reveals the quality of a compiler. Programs can be generated which are several orders of magnitude more complex than users would write. If such programs are handled correctly by a compiler, one can have great confidence in the product provided the generator covers the necessary language features. One compiler included an artificial limit on the complexity of expressions that could be handled which reduced the ability of the test generator to test the product — fortunately, this approach is rarely used and compilers are typically good at handling complex expressions. Of course, it is quite acceptable for compilers to reject very complex expressions; although this could cause problems if an expression has been generated by a symbolic algebra package (for instance) in order to solve a real application (as opposed to stress-testing).

The testing of the compiler for a Pascal subset for AWE is of interest in its own

right, since the complete compiler was formally specified in Z and the implementation derived from that, but written in Prolog. In spite of this highly rigorous development, additional testing was thought worthwhile since the compiler has only one user. A pre-release of the compiler was checked by means of the generator which revealed two bugs: in the operator **mod**, and 2-dimensional array access. Unlike many parts of the compiler, these parts had not been subject to formal proof.

The experience so far with stress-testing compilers by NPL is summarised below.

Language	*Compiler*	*Test*	*Results*
Pascal	5 unnamed	Informal, 1988	4/5 had safety bugs
Pascal	For a PC	Informal, 1996	safety bugs found
Pascal	For Unix	Informal, 1996	safety bugs found
Pascal subset	pre-release, AWE	Formal, 1996	two safety bugs
CHILL	pre-release, Unix	Informal, 1994	major defects (now corrected)
Haskell	Glasgow, unopt	Informal, 1992	safety bugs
Ada	Telesoft 386	Formal, 1993	minor defects
Ada	Alsys, Transputer V5.4.8	Formal, 1994	minor defects/ capacity defects
Ada	Alsys RISCAda V3	Formal, 1995	minor defects
Ada	AONIX Sun4 V5.5.2	Formal, 1998	two safety bugs

Table 1: Summary of compiler stress-testing

In the table, we mean:

(in)formal testing. Formal testing implies an extensive set of tests undertaken in a repeatable and reproducible manner with a test report.

safety bug. Compiling a correct program incorrectly without warning. This is detected by the failure of an internal check on execution.

minor defects. Failing to compile a correct program containing an unusual construct.

capacity defect. Failing to compile programs of modest size and complexity.

major defects. Safety bugs plus other defects so that few programs are apparently handled correctly. (The CHILL compiler has since been extensively tested and has been successfully used for critical projects.)

The conclusion of this experience is that this form of compiler stress testing is effective. Unfortunately, the approach is not widely adopted so that several languages have validation suites but no stress testing capability. Producing tests which cannot be checked by execution does not seem worthwhile, since this only exercises the front-end of the compiler. Applying stress-testing very early in the development of a compiler is also not effective, since too many programs are rejected. The other problem with the method is that the generated programs typically report the same error many times, potentially hiding other errors. Of course, this is only a problem when errors have been detected.

3 Component Testing

The application of randomly constructed test sets to software components would appear to offer the same benefits as for compilers. Hence the author considered the implications of adding random testing to the British Computer Society component testing standard [8].

The use of such testing is certainly accepted in the sense that published material uses the method. For instance, random argument values are computed for the application of tests of the standard mathematical functions (sin, cos etc) in the Pascal Validation Suite [3]. However, the use of this technique is effective due to the ease with which automatic acceptance checking can be applied. With such a process, a large number of tests can be run automatically. Hand analysis of the few that fail (if any) can then be undertaken. According to the strength of the acceptance logic, the testing can be very thorough. Even when the acceptance criterion is merely that the program does not crash, confidence in this property is obtained at modest cost.

The strength of conventional component testing is assessed by metrics such as statement coverage. With random testing, the number of random tests is of little value unless the distribution is a reasonable fraction of the entire input domain. This presents a problem with the BCS standard, since a key aspect of the standard is the testedness metric. The approach taken below is that the coverage of the input domain is measured by means of the equivalence partitions used for equivalent partition testing, together with the number of tests run.

The paper of Thévenod-Fosse et al[10], gives strong support for the use of random testing. Moreover, the method of biasing the random numbers to include powers of two and end-points more frequently (which is used in the NPL compiler stress-testers) is also supported.

The appendix to this note contains the proposal made to the BCS component testing group. The illustration is a simplified version of the testing undertaken in the Pascal Validation Suite for the square root function.

4 Conclusions

The author thinks that random testing is a test method which is under-exploited in practice.

5 Acknowledgements

Details from the Haskell stress-tester was provided by Nick North. Steve Austin undertook the formal testing of the three Ada compilers. I am grateful to AWE plc and Alcatel for permission to publish the testing of the Pascal subset and CHILL compilers, respectively.

References

[1] F Bazzichi and I Spadafora. An automatic generator for compiler testing. IEEE Trans on Software Engineering SE-8. 1982, pp343-353.

[2] D L Bird and C U Munoz. Automatic generation of random self-checking test cases. IBM Systems Journal 1983 pp229-245.

[3] B A Wichmann and Z J Ciechanowicz (editors). Pascal Compiler Validation. Wiley. 1983.

[4] K V Hanford. Automatic generation of test cases. IBM Systems Journal. 1970. pp242-257.

[5] B A Wichmann and I D Hill. An efficient and portable pseudo-random number generator. Applied Stats. 31. 1982. pp118-190.

[6] S M Austin, D R Wilkins and B A Wichmann. An Ada Program Test Generator. TriAda Conference Proceedings. ACM. October 1991.

[7] The Ada Compiler Validation Capability. Details available on the Internet: http://www.informatik.uni-stuttgart.de/ifi/ps/ada-software/html/evaluation.html

[8] British Computer Society Specialist Group in Software Testing. Standard for Software Component Testing (Working Draft 3.3). Glossary of terms used in software testing (Working Draft 6.2). April 1997.

[9] B A Wichmann and M Davies. Experience with a compiler testing tool. NPL Report DITC 138/89. March 1989.

[10] P Thévenod-Fosse, H Waeselynck and Y Crouzet. Software Structural Testing: An Evaluation of the Efficiency of Deterministic and Random Test Data. Predictably Dependable Computing Systems Report No 57. December 1991.

A Component Testing: an addition for random test cases

This appendix is a proposal, first drafted for discussion by the SIGTEST Working Party, for the addition of Random Testing to the existing component testing standard [8] (version 3.0).

It may be possible to use this as an example of how proposals for additional techniques should be prepared.

A.1 Addition/Changes to Clause 3

Add:

3.13 Random Testing

3.13.1 Analysis

Random testing uses a model of the component that partitions the input values of the component. These partitions may the same as that used for equivalence partitioning (see 3.1.1).

The model shall contain input values. Both valid and invalid values are partitioned in this way.

3.13.2 Design

Test cases shall be designed to exercise partitions. The input values for each test case within a partition shall be constructed using a repeatable random process.

For each partition, the following shall be specified:

- The distribution function of the random input values.
- The number of random values used (and their values or the specification of the process used for their construction).

Test cases are designed to exercise partitions of valid and invalid input values.

A.2 Addition/Changes to Clause 4

Add:

4.13 Random Testing

4.13.1 Coverage Items

Coverage items are the partitions described in the model (see 3.13.1).

4.13.2 Coverage Calculation

Two calculations are defined: partition coverage (see 4.1.2) and the number of test cases used.

A.3 Addition/Changes to Annex B

Add:

B.13 Random Testing

Introduction

Random testing is a black-box technique and hence is useful when no information on the internal structure of the software can be used. Clause 3 is written so that the random values could be determined manually as well as by use of a pseudo-random number generator. To ensure the tests are repeatable, it is not acceptable to use a pseudo-random number generator which cannot be re-run to produce the same values (when the test values have not been recorded).

In practice, the use of the technique is most effective when the output from the result of each test can be automatically checked. In this situation, very many tests can be run without manual intervention.

In ideal circumstances, it may be possible to derive some reliability data from the result of random testing if it can be shown that the distribution used corresponds to that which would arise in actual use of the component.

Example

Consider a component, $sqrt$, whose specification is as follows:

> The function $sqrt$ has a single floating point parameter, x and produces a single floating point result, y. If $x > 0$, then $(1 - \epsilon)x \leq y^2 \leq (1 + \epsilon)x$, where $\epsilon = 10^{-9}$.

In this case, we have a simple means to determine that the result of the function is correct, according to the above specification. Hence random testing is a reasonable technique to apply.

The analysis of the above specification reveals three input partitions: $x = 0$, $x < 0$, and $x > 0$. The only partition for which random testing is useful is the valid partition containing many values, namely the positive values.

The random values for x used in the test are taken from a uniform distribution in the range 1.0 to 2.0. The complete test program to undertake this in Pascal is:

```
program rtest(output);

const
   Ntests = 10000; { Number of random tests to perform }
   eps = 1.0E-9;
var
   ix, iy, iz: integer;
   { Seeds for the pseudo-random number generator }

   count: integer;
   x, y: real;

procedure RandomStart;
   begin
   ix := 1; iy := 2; iz := 3;
   end;

function Random: real;
   { Returns random value 0.0 <= Random <= 1.0 }
   var
      rndm : real;
   begin
   ix := 171 * ix mod 30269;
   iy := 172 * iy mod 30307;
   iz := 170 * iz mod 30323;
   rndm := ix/30269.0 + iy/30307.0 + iz/30323.0;
   Random := rndm - trunc(rndm)
   end;

begin
RandomStart;
for count := 1 to Ntests do
   begin
   x := 1.0 + Random;
   y := sqrt(x);
   if ((1.0-eps)*x > y*y) or (y*y > (1.0+eps)*x) then
      writeln('Tests failed: x, sqrt(x), number',
x, y, count);
   end;
writeln(Ntests:1, ' random tests executed');
end.
```

Notes

This method of testing is that used in the Pascal Validation Suite to check all the standard functions. The 10,000 tests in this example only take a few seconds to execute.

The usual method to implement the square root function implies that using just the range from 1.0 to 2.0 is not a significant defect.

A.4 Addition/Changes to Glossary

Add:

> **random testing:** A *test case design technique* which uses input values for a specified input *equivalence class*. The input values are random values from a specified distribution.

A.5 Version of Random Testing now in the BCS standard

It was agreed that random testing as a method should be added to the BCS component testing standard, but some differences were made from that presented here as follows:

1. The concept of using the equivalence partitions above was removed in favour of considering all input values.
2. The proposal here explicitly mentions invalid input values, while the agreed version is silent on this issue.
3. The input distribution is either based upon the known actual distribution or a uniform distribution (rather than allowing any defined distribution as above).
4. The BCS standard does not define any coverage measure.
5. The BCS standard has a different example, for which uniform distribution over the entire set of input values makes better sense.
6. The Glossary (version 6.2) does not define random testing.