Empir Software Eng (2006) 11: 5–31
DOI 10.1007/s10664-006-5964-9

SPECIAL ISSUE PAPER

The Economics of Unit Testing

Michael Ellims · James Bridges · Darrel C. Ince

Editors: Antonia Bertolino and Eda Marchetti

Abstract Conventional wisdom and anecdote suggests that testing takes between 30 to 50% of a project's effort. However testing is not a monolithic activity as it consists of a number of different phases such as unit testing, integration testing and finally system and acceptance test.

Unit testing has received a lot of criticism in terms of the amount of time that it is perceived to take and its perceived costs. However it still remains an important verification activity being an effective means to test individual software components for boundary value behavior and ensure that all code has been exercised adequately. We examine the available data from three safety-related, industrial software projects that have made use of unit testing. Using this information we argue that the perceived costs of unit testing may be exaggerated and that the likely benefits in terms of defect detection are quite high in relation to those costs.

We also discuss the different issues that have been found applying the technique at different phases of the development and using different methods to generate those tests. We also compare results we have obtained with empirical results from the literature and highlight some possible weakness of research in this area.

Keywords Software testing · Unit testing · Adequacy criteria · Test effectiveness

1. Introduction

The general computing literature tells us that "testing" as an activity can take up 30 to 50 percent of the total effort consumed by a software development project. Values for test effort as a percentage of total effort from a number of standard texts are presented in Table 1.

However testing is not a monolithic activity; it is usually discussed in terms of the types of testing activity being performed; examples being unit testing, module

M. Ellims (✉) · J. Bridges
Pi Technology, Cambridge, UK
e-mail: mike.ellims@pitechnology.com

D.C. Ince
Open University, Milton Keynes, UK

Table 1 Test effort data from standard texts

Reference	% Given
Brookes (1995)	50
Shooman (1988)	30–50
Pressman (2000)	30–40
Sommerville (2004)	50

testing, integration testing, system testing and so on. While overall figures are often given in the literature, the authors can find no work that gives reliable estimates for the various component activities. This paper attempts to rectify this situation.

This paper details some of the results obtained from unit testing activities on three different industrial projects and in a real-time area that can be broadly classed as "automotive." The primary focus of the paper is to examine the unit testing activity; however, where relevant, we put this activity into the context of the other verification and validation activities (particularly testing) performed on the software.

The three projects examined are described in detail in Section 4: Wallace is an engine control system of approximately 13,500 lines of executable code[1] (LOC), Grommet is a set of modules that comprise part of an engine control unit of approximately 10,700 LOC and Sean is a "smart" sensor comprising around 3,500 LOC.

The three projects have a number of factors in common. First, they were all undertaken by the same company; second, they all had as a basis similar quality plans based on the same generic company quality system. However it should be noted that the generic quality system allows a large degree of freedom in the exact form that the specific project-based quality plans can take (Ellims and Jackson, 2000). Thirdly, all projects made extensive use of bench or system testing (Aitchison and Goody, 2000; Ellims, 2000), even building specific test equipment where necessary (Ellims et al., 2005). Importantly they all also used a unit test tool derived from a common basis (Ellims and Parkins, 1999). Finally some of the personnel involved in all three projects were the same, one of the authors being employed as the project architect on two of the systems (Wallace and Sean) and was involved in formulating the unit test requirements for the other project (Grommet).

Section 2 examines previous work in this area. Section 3 provides definitions, process framework and discusses the data sources utilised for this work. Section 4 of the paper gives an overview of each project and specific details of the development process followed. Section 5 looks at what was found during the unit testing for each of the projects. Section 6 contrasts the results from the three projects and Section 7 discusses the major features of what was observed and in Section 8 we present our conclusions. We finish in Section 9 by discussing some other analysis activities we could perform with the available data.

[1] In this paper the term lines of code always refers to executable lines, including other referenced work.

2. Previous Work

Unit testing is now, or at least should be, an integral part of safety related software projects. While much has been written on the theoretical aspects of unit testing, for example test adequacy criteria such as Frankl and Weyuker (1983), Parrish and Zweben (1991), Richardson and Thompson (1993), Zhu (1996) and of metrics for software testing (McCabe, 1976; Davis and Weyuker, 1988) there have not been very many published results from real software projects.[2] For example, in a comparatively recent major review paper on software test adequacy (Zhu et al., 1997) few citations from over 220 were to statistically valid empirical studies.

The reasons for this state of affairs are clear: industrial staff rarely have the time to analyze past projects before being moved to other projects and academics very rarely have access to statistically valid collections of data.[3] Consequently software developers have had to rely on anecdote, myth and software engineering textbooks. A criticism strongly echoed by Fenton and Ohisson (2000) and Tichy et al. (1995) among others.

There have been a limited number of studies which have drawn on empirical data from humans performing test activities. Some of the better (in statistical terms) are outlined below.

Myers (1978) undertook a study using 59 experienced engineers as subjects, though all were not programmers they are all described as being "better than average." The experiment consisted of evaluating functional and structural testing using individuals and walkthroughs/inspections using teams of three.

Basili and Selby (1987) in their classic comparative study of validation methods used 74 subjects and four relatively small programs to examine the effectiveness of code reading, functional testing using equivalence partitioning and boundary value analysis, and structural testing using 100% statement coverage as the completion criteria.

Kamsties and Lott (1995) performed a study to replicate that undertaken by Basili and Selby (1987) using a fractional-factorial design, observing the effect of functional, structural and inspection techniques over three programs. Wood et al. (1997) repeated this experiment as part of a software engineering course.

Laitenberger (1998) carried out a study in which 20 graduate students carried out code inspections and structural testing of 262 executable lines of C code. The two main findings were that inspections were superior to structural testing and that inspections and structural testing do not complement each other well.

So et al. (2002) performed a large scale exercise with 34 students divided into teams of four over 12 weeks as part of a course in software engineering. The aim of the associated research was to determine which of a set of six defect detection techniques was the most effective. The conclusions they came to were that N-

[2] A set of Google searches involving keywords such as 'empirical,' 'unit testing,' 'industrial' 'data,' 'metric,' 'coverage,' 'experiment' and 'structural' returned hardly any hits at all. Searches using CiteSeer were only marginally better.

[3] One of the few large scale ongoing empirical-based projects the NASA-backed Software Engineering laboratory at the University of Maryland closed down fairly recently depriving empirical researchers of probably the only major source of realistic project data. This project produced the vast bulk of useful statistical data for researchers over the past 15 years including Ince and Shepperd (1993).

version programming, testing and Fagan inspections were the most effective fault detection techniques of those examined.

Maximilien and Williams (2003) report on the use of a test-driven approach reliant on extreme programming concepts on a project which reduced the defect rate at IBM by a factor of 50% over *ad-hoc* unit testing.

Lyu et al. (2003) carried out an experiment in which they measured the effectiveness of coverage testing vs. mutation testing using 34 independent versions of an avionics program developed by computing students and came to the conclusion that while structural coverage was an effective testing method its use as a quantitative indicator of testing quality was less good than mutation measures.

Runeson and Andrews (2003) compared white-box testing and code inspection, examining the results of these two activities carried out by graduate students. The result of this experiment was that testing detected more errors than inspections, but that the latter was more efficient in terms of time. Moreover it was found that inspection was more efficient in isolating the cause of an error.

Torkar et al. (2003) examined a large open source component which had supposedly been thoroughly tested and which was in use and discovered 25 faults even though it had been thoroughly functionally tested. They considered that a number of these faults were so serious that they would have given rise to major errors in any system using the component.

Unfortunately the majority of these studies suffer from two major problems, that of size and realism. In the case of size, either the systems under investigation were too small or only a small part of a larger system is studied. As concerns realism, too often the subjects of the research are university students (Ince and Shepperd, 1993). In this paper we report on results from three large industrial projects.

One final interesting piece of work was conducted by Dupuy and Leveson (2000) investigates the effectiveness and cost associated with meeting MC/DC coverage criteria (as per the DO178B standard, Anon., 1992) on the HETE-2 Satellite software after functional testing had been performed and is similar to work undertaken on the Grommet and Sean projects.

Given the limited amount of empirical data that is available there are still many important research questions that at best are only partially answered or at worst not answered at all. These include:

- How efficient is unit testing as compared with other verification activities such as code reviews?
- What resources does unit testing consume on software projects as a proportion of overall spend?
- Are the resources consumed on unit testing different in different types of projects?
- How effective are structural coverage metrics such as coverage of LOC or coverage of branches at determining test effectiveness?

This paper describes some large-scale, empirical data on testing which we hope will help researchers make the first steps to answering some of these questions and help illuminate some of the views that have been put forward with little experimental validation, for example Hamlet's (1994) view that it is misleading to assume that trustworthiness and reliability can be demonstrated via coverage metrics and Garg's (1994) almost contrary view that software reliability metrics are intimately related to software dependability.

In addition to the above, while reworking the original paper (Ellims et al., 2004) we decided to compared the results from previous empirical work surveyed above with results presented here to see what comparisons could be made.

3. Definitions, Process and Data

3.1. Definitions

Unit test is the testing of a single C function in isolation, using a test harness to provide inputs to the function under test, collect and then check the outputs produced. Called functions are normally replaced by test stubs to provide complete control. All the unit tests described in this paper were performed using a version of the in house Test Harness Generator (THG). The THG tool (Ellims and Parkins, 1999) uses a spreadsheet as its user interface with users manually defining input and output fields. The advantage of a spreadsheet format is that it puts at the users' disposal all the inbuilt facilities for generating the expected outputs which is excessively laborious if done manually. A macro program provided as part of THG converts the spreadsheet data into a standalone test harness thus removing another major overhead (Dalal and Mallows, 1998) of supplying the test scaffolding. Note that this definition says nothing about whether the tests are structural or functional and both are applicable and encouraged (see Appendix for details).

There are limits to how spreadsheets may be used, the use of the built-in programming language (Visual Basic for Applications, VBA) for generating expected results is forbidden. Expected results are required to be calculated using normal spreadsheet facilities. This is done for two reasons, firstly allowing use of VBA would encourage cut and paste conversions removing most of the independence between code and unit test. Secondly forcing the use of the spreadsheet means that where necessary a functional programming paradigm is being employed. Green (1990) observes that the spreadsheet is a declarative programming environment (as opposed to imperative language such as C) and that, "it hardly counts as programming."

Module test is the testing of groups of related functions, usually in a single file. For example, on Wallace there is a module for analogue input processing, a module for analogue output processing etc. During module test, called functions are not normally replaced by test stubs so the complete call tree is activated. While THG is used for almost all unit test activities within Pi Technology, a larger variety of test tools are used for module testing. THG can be used but it is problematic to define all the variables that may be involved though many can be set and left at default values. Other methods include compiling the code as part of a complete system and using a debugger with breakpoints and building bespoke test harnesses; however the latter is expensive in terms of time expended.

Integration test is testing to ensure that interfaces to code external to the module being integrated with the rest of the system are correct. Techniques used are similar to module testing and system test techniques can also be applied. Usually there is a nontrivial overlap between the module and integration test activities.

System testing is the process where by the complete system comprising software, hardware and often sensors and actuators are combined together with a hardware in the loop (HIL) test environment to provide the model of the plant under control and to provide generation of inputs (e.g., analog and digital inputs) and check outputs for accuracy and timing. Testing is performed either using scripted tests or in many cases by manually executing tests. Details of the system test environment can be found in (Aitchison and Goody, 2000; Ellims, 2000).

Prototyping is the technique of circumventing the majority of the process to directly address the question "are we building the right thing?" irrespective of whether we have built it right. This was necessary for a small number of modules where insufficient domain knowledge existed to give confidence that the functional specifications were adequate. Prototyping is effectively an ad hoc system testing activity. It should be noted that once prototyping has been completed the full development process would always be applied.

3.2. Process

The generic quality management system (QMS) at Pi Technology is based on a standard "V" model (McDermid and Rook, 1991) though the QMS has been specifically set up so that it is simple to modify for specific projects (Ellims and Jackson, 2000) to give us as much flexibility as possible. The use of this flexibility is demonstrated in both the Grommet and Sean projects. The QMS itself meets the requirements of ISO 9001 and has achieved ISO 15504 Capability Level 3 for all processes in the defined scope of assessment that cover the requirements of the key European car manufacturers. The specific Wallace quality plan and activities has been externally audited to be consistent with the requirements for software to be developed for DO178B (Anon., 1992) Class B applications with only small changes needed to meet Class A requirements.

The left side of the "V" is comprised of the following major steps; requirements analysis and definition, functional requirements, architectural design, module design and coding. The right side of the "V" contains the corresponding verification and validation activities.

Each of the major components listed above consists of the number of sub-components such as review activities and checklists. Of particular interest here is the process for unit testing. Figure 1 shows the unit test activity and the simplified process flow within the activity. The term *simplified* is used as not all features are shown, for example the activity Code Correction or Update involves a change request (CR) being raised, approved and acted on.

It should be noted that there are two phases to test execution, one on the host and one on the target system. Host execution is allowed and even encouraged as running tests on the host is far more efficient than on the target because of better tools (debuggers) and faster turnaround. However the tests have not passed until they have been executed successfully on the target hardware as issues with processor hardware and compilers are not that uncommon.

3.3. Data Sources

A number of data sources have been used to provide the data presented here, all of which were constructed during the normal course of the project work.

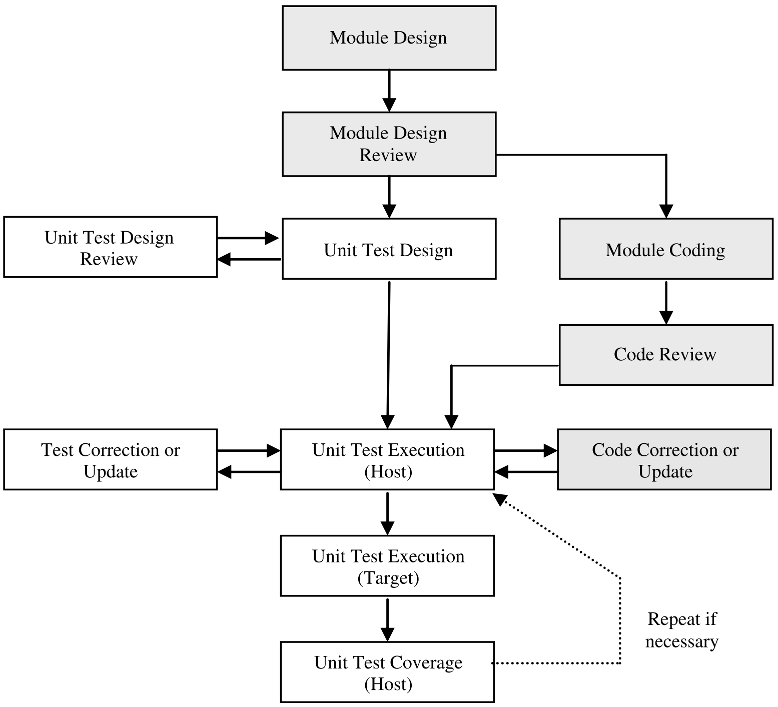

Module Design
Module Design Review
Unit Test Design Review
Unit Test Design
Module Coding
Code Review
Test Correction or Update
Unit Test Execution (Host)
Code Correction or Update
Unit Test Execution (Target)
Repeat if necessary
Unit Test Coverage (Host)

Fig. 1 Simplified process for performing unit tests, shaded boxes show associated activities that must be completed before or in conjunction with unit testing

Timesheets are spreadsheets with project and sub-projects codes, the sub-project code defining what activity and on Wallace what module is being worked on. These are filled in weekly and reflect the actual amount of time spent on each activity.

Review records are text documents that record the results of each review and where up or down stream work products need modification, along with a reference to the change requests used to track any necessary changes in those work items.

Change requests document all errors, changes in functionally and/or implementation and requests for new functionality. Information is recorded in an MS Access database to simplify location and tracking of work products that need to be changed. All required changes not discovered as part of a review have change requests associated with them. The CR system acts as the "collective memory" for the project.

Code coverage, on the Wallace project this was measured using LDRA TestBed tool and on Grommet and Sean using the GCT freeware tool.

4. The Projects

4.1. Wallace

Wallace was a full function engine control system; it comprises modules to perform hardware level I/O on discrete analogue and digital lines as well as dealing with a number of different communications channels. Its primary function is to control the performance and power output of large industrial internal combustion engines via spark timing, digital knock detection, air fuel ratio control etc.

It was a fairly classic project in that it more less follows the standard project life cycle based on the V model (McDermid and Rook, 1991). Therefore, for most intents and purposes, detailed design followed specification, coding followed detailed design, unit testing followed coding, integration testing followed unit tests and hardware in the loop (HIL) (function or bench testing) followed integration testing. At this point you fly to the USA to perform the combined system/acceptance test which involved endless discussion about whether any fault observed is in the loom, an actuator, sensor or any other item that can fail. This is an extremely expensive exercise both in terms of personnel, but also in costs associated with operating the test facilities.

The major work products associated with the unit test phase were the detailed design, the unit test construction and analysis of the results of running those tests. The detailed design was a text document that consisted of a high level overview of the module, an extracted version of the data dictionary and pseudo-code for each of the functions that comprise the module. Fixed point code (the 32 bit target processor does not support floating point) was generated by hand from the designs.

Other test activities are a combined module/integration test phase and extensive system/bench testing of the complete system using a hardware in the loop simulation system and either the various external devices being controlled (i.e., sensors and actuators) or digital models of those devices.

4.2. Grommet

Grommet was a project to provide new functionality for a hybrid power train control system, it comprised modules to provide the following functionality:

- input signal processing,
- failure detection and mitigation,
- vehicle security,
- power train co-ordination.

Grommet was also a fairly classic project in that it more or less followed a standard project life cycle based on the V model, the main difference being that the unit test activity was allocated to the end of the development process. Detailed design followed specification and coding followed detailed design. Module testing of functional behavior followed coding, integration testing, concluding with bench and vehicle-based testing by the customer. Unit testing was deferred until the end of the project because the software requirements and software designs were in a continual state of flux. The intention here was to remove the cost of maintaining unit test cases during development. It should however be noted that this was *not* ideal but was a reasonable response to the serious problem of requirements churn, on Sean doing this proved to be (at least in part) counter productive.

The major work products associated with the unit test phase were the detailed design, the unit tests, the results of running those tests and associated coverage metrics. The detailed design itself was an executable model-based design with a text document that consisted of a high level overview of the module and an extracted version of the relevant data dictionary information. Floating point code was generated by hand from the designs.

The unit tests comprised a number of sheets in a spreadsheet workbook from which a C language test harness was automatically generated using the same in-house tool, THG. There were two outputs from the test: the test results themselves and secondly the coverage data collected while running the tests.

Other test activities involved a module/integration test phase and extensive system/bench testing of the complete system using hardware-in-the-loop simulation. One significant difference between this project and Wallace is the unit test data creation activity was out-sourced to a company in India with a very tight specification (Appendix).

4.3. Sean

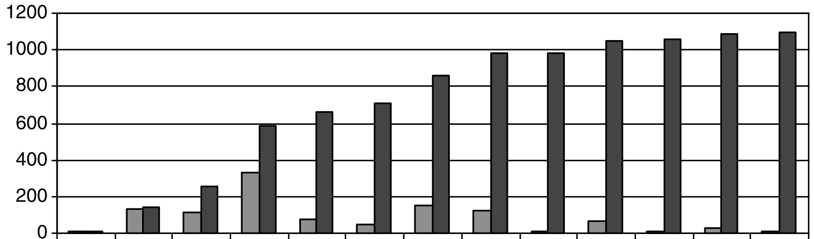
Sean was a smart voltage and temperature sensor which processed a large number of inputs, performed a small amount of filtering and error detection on those values and then transmitted them to a master control ECU over a communications link. Unlike Wallace and Grommet, Sean was a crash program to independently re-engineer the software to meet process requirements for safety related software. Thus it was required to possess all the attributes of a process similar to Wallace and Grommet but within a compressed time scale.

To meet the delivery schedules for the first release of software the process was modified by moving the unit testing phase to the end of the process so that it was effectively the last activity performed. To compensate, more effort was put into the bench/system test phase of the process, which was started while the requirements were still being developed. This allowed special purpose hardware to be developed to aid that testing activity. This strategy was helped by the fact that the functional requirements document contained specification detail down to a very low level, so testing against requirements would actually check the implementation to a large extent.

The process outputs were very similar to the Wallace project in that the requirements, software architecture, and detailed design documents were all text documents supported by a data dictionary; again fixed point code was developed from the designs. However unlike Wallace the in-house requirements tracking tool PixRef[4] was used to trace each numbered requirements though each work product, so an exact correspondence between requirements and functional tests could be made automatically.

Like Grommet, large parts of the work were out-sourced to various companies in Russia and India, though all design work was kept in house. In particular the initial generation of unit test data was performed offshore to a re-worked and cut-down version of the test data generation specification used on the Grommet project, for details see Appendix.

[4] The Pixref tool will be available for download from http://www.pitechnology.com/.



	fs	fs rev	des	des rev	code	code rev	ut des	ut run	int des	int run	sys des	sys run	engine
▨ correction	10	133	109	332	78	46	149	123	7	64	7	24	13
■ cumulative	10	143	252	584	662	708	857	980	987	1051	1058	1082	1095

Fig. 2 Corrections found at each phase and cumulative totals

5. Test Results

5.1. Wallace

Unit testing on Wallace had the requirement that the tests should be branch adequate, normal testing heuristics were applied where they seemed applicable. For example boundary values analysis was applied to all values that had physical interpretations as this seems effective (Jorgensen, 1995)—however in this context it means most variables. Heuristics such as checking loops for zero, one and two iterations were not applied as there were no loops which were not statically bound.

Analysis of Wallace fault data was originally performed in 1999 by manually reviewing all change requests and review records to count the number of changes which were just corrections (excluding "improvements") to each work item recorded. The major results are shown in Fig. 2. This shows the total and cumulative changes for each phase of work. The activities shown left to right are as follows; *fs*—functional specification, *fs rev*—review of functional specification, *des*—detailed design, *des rev*—design review, *code*(ing), *code rev*—code review, *ut des*—unit test design, *ut run*—unit test execution, *int des*—integration test design, *int run*—integration test run, *sys des*—system test design, *sys run*—system test run and *engine*—on engine testing.

From the development point of view the most pleasing feature is the shape of the cumulative curve which clearly shows that the rate of change is dropping. Data from field use indicates that this trend may accurately reflect the number of actual errors removed. Conservative estimates from the customer indicate that in excess of two million operating hours have been accumulated by the software in the field with no software related errors having been reported.[5]

[5] The engines being controlled by the Wallace software are in operation 24/7 for periods of up to six months using techniques such as hot oil changes and spark plug replacement while in operation. One confounding factors that needs to be taken into account is that since results reported here were produced the software has been under maintenance and had significant new functionality added over last five years. This is an on going process.

From Fig. 2 it can be seen that the most effective error detection strategies in terms of finding errors in the system were design review, unit test design, review of the functional specification and the unit test runs. In total these activities account for around 60% of the necessary corrections. One interesting feature of the data above is that more errors were located during the design of the unit test data than were discovered during the execution of the tests themselves. This is probably because the unit test design activity itself forces the unit test designer to examine what they were attempting to achieve in a new and rigorous manner. Effectively the unit test design process is a type of review.

Note that the test design is effectively a functional programming process, how significant the use of a declarative environment may actually be in practice is difficult to gauge, however we would propose that this is at least in part the cause for the high rate of detection during the test design phase.

It is also possible to attach costs in terms of time to each of the activities by extracting data from time sheets as shown in Table 2. This shows the activities of design review (DES REV), unit test design (UT DES), functional requirements review (FS REV) unit test run (UT RUN) as a percentage of total engineer hours. For comparison the total figures for integration (INT TEST) and system/bench testing (SYS TEST) are also given. Percentage time expended on unit test activities compare favorably with the total effort expended on integration and system test activities.

It is instructive to examine the data in terms of cost in hours per correction made. For example the detection rate (in hours expended) for the unit test activities is 3.6 and 0.6 hours per correction giving an average value of 4.2 hours per defect. If each of these corrections were not discovered until integration testing was performed then the potential correction cost goes up by nearly a factor of five. For system testing the cost would rise by a factor of two. This can primarily be attributed to the additional effort required to perform these testing activities in terms of difficulty finding the error and rerunning the tests. However, it should be noted there is no guarantee that all the changes made due to unit testing would be found by those activities (and vice versa).

It should be noted, however, that the figures for system testing given in Table 2 are, in one sense, an aberration as much work that could potentially be classed as system test has been recorded as prototyping activity. If this is included with the system test figures then the time increases to 14% of the effort and the hours per detection increases to 50 and the difference in the ratios increases to a factor of 12.

5.2. Grommet

The total number of defects located by unit testing is given in Table 3. This shows the total number of code defects discovered during unit test design (out sourced),

Table 2 Percentage time and detection rates for each activity

Activity	DES REV	UT DES	FS REV	UT RUN	INT TEST	SYS TEST
% TOTAL time	1.7%	3.7%	0.6%	0.5%	10.7%	2.5%
Hours/change	0.8	3.6	0.6	0.6	21.7	11.6

Table 3 Number of defects detected on Grommet

Activity	Number of errors found
Test design	25
Run on host	32
Run on target	0

run of the unit tests in the host environment (windows PC) and re-running on the target environment. Unlike the Wallace data above more errors were found during the execution of the test cases rather than the unit test design stage. This may be a factor with outsourcing the creation of unit test cases.

Unfortunately, on this project (unlike Sean) we were unable to classify errors as either minor or major, where a minor error did not require code to be modified but major errors did (see Section 5.3). This was because it is not always easy to identify the system severity of the effect of failure. In many cases it is less time consuming to fix an issue raised at unit test rather than to judge the true impact. This was possible on the Grommet project because, even though unit testing was deferred to relatively late in the process, there was still sufficient project time that the risk of introducing defects through late changes was considered acceptable. It should be noted that this was not necessary on Wallace due to looser time constraints.

It is unusual that unit testing was deferred in this manner. However, there were a number of reasons for doing this. First, the requirements were still in a state of flux at the point where unit testing would normally be performed; thus there was significant risk that the work would have to be redone almost completely. Second, it was felt at the time that the functional module testing would be capable of discovering the majority of the errors. Given the number of errors found during unit testing the original presumption does not now seem reasonable.

In terms of effort to detect a defect using unit testing required an average of 8.4 days per defect. While this may seem excessive it must be remembered that unit testing occurred at the end of the project after several iterations of the software which included the complete life-cycle though to acceptance test on a vehicle. Thus the majority of errors should have already have been found though review, module, system, track and on road testing. The surprise for the development team was that so many errors remained. Overall unit testing required 10.8% of the total project effort and through outsourcing resulted in costing the customer 5% of the total project costs.

What we are able to conclude from the data above is that 57 code defects escaped from all other verification and validation activities. As stated above this was contrary to what was expected due to having had a fairly high level of confidence in the rigor of the module functional testing. Due to the nature of the module testing performed on Grommet, it was possible to gather coverage metrics usually used to determine whether unit test cases are sufficient, namely statement and branch coverage. Table 4 compares the coverage metrics for unit testing and module testing which compares favorably with figures reported in the literature (Dunietz et al., 1997; Burr and Young, 1998).

It should be noted that during the module functional testing it was not the intention to obtain full statement or branch coverage on code, that is code coverage adequacy was not the formal completion criteria. Rather 100% requirements

Table 4 Statement and branch coverage metrics for unit and module testing on Grommet

Coverage	Module test	Unit test
Mean statement coverage	96.6%	99.7%
Mean branch coverage	82.5%	98.9%

coverage was required. Tests were "adequate" if they demonstrated that the functional behavior of the code relative to the executable specification was correct by executing both with the same test vectors. It should also be noted that it would probably not have taken a great deal more effort to increase the coverage of the functional module testing if statement coverage was to be included as a criteria for test completeness. However, what extra utility this would have provided is not clear from the data available given that only 3.4% of the code was not executed.

5.3. Sean

Like Grommet the Sean project performed its unit test activity as one of the last process activities. In practice the moving of unit tests to the end of the process did not significantly affect the development of the majority of the modules. However for the largest of the modules we faced significant issues due to its relatively complex logic and the fact that it performed a significant amount of fixed point numerical computation. Though additional functionality to increase visibility of internal data values (Freedman, 1991; Voas and Miller, 1995) had been specifically added during requirements development, this proved in some notable cases to be insufficient.

In the functional test environment establishing what code was being run when an erroneous output was observed was difficult to determine as it was not possible to step though the code. In the most extreme instance this forced the engineers responsible for the particular module being tested to build an *ad-hoc* unit test environment where they had sufficient control of the code to be better able to design specific tests to probe the code. Several significant errors were found this way, of particular interest was that one of the errors located was found to be a compiler bug. In the *ad-hoc* environment the error was found not to occur; examination of the generated assembler code for the target revealed the cause of the problem (compiler error) and the "C" program logic was rearranged to ensure that generated target code was correct.

It is also interesting to note that the engineer with primary responsibility for this module commented (forcefully) that it should have been unit tested before we attempted to perform functional testing.[6] We discuss this change in attitude further in Section 7.

The total number of defects located by unit testing is given in Table 5 which shows the total number of code defects discovered during test design (out-sourced), running the unit tests in the host environment (windows PC) and re-running on the target environment. As for the Wallace data, more errors were found during the test data design process than during the running of the tests. Unlike the Grommet project, on this project we were able to classify errors as either minor or major,

[6] The same engineer also worked on the Grommet project and had expressed doubts as the utility of performing the unit testing!

Table 5 Total (and major errors) in code for all modules

Activity	Number of errors found
Designing tests	36 (3)
Run on host	8 (5)
Run on target	1

where a minor error i.e., those that did not significantly affect the safety or functionality of the system after analysis did not require code to be modified, but major errors did. Data for major errors is shown alongside the totals in Table 5 in parentheses. Examples of minor errors include the following:

- Access off the end of an array—system would be non-functional long before condition can be met in real world therefore modification is not required on safety grounds and not required on functional grounds.
- Temperature comparison uses ">" rather than ">=," given the lag in the physical system and other physical limitations a difference of 1/10 degree Celsius was not significant and does not affect safety.

A major error that was corrected was an instance of a communication buffer overrun as this could be expected to occur in the field due to electrical noise. The change was allowed as system analysis showed there would be no safety issues due to hardware redundancy for critical voltage measurements. Classification of errors as major/minor is not an ad-hoc process, in the first instance it requires agreement between the engineer responsible for the software module and the project architect and then agreement between the project architect and senor engineers from the customer and the vehicle program. All errors not corrected at this point remain in the change request system and it is intended to correct these in any future release of the software.

A notable feature of the data given in Table 5 is that running the unit tests found more of the major errors than the test design process, but overall the design process discovered numerically more errors.

The reasons for the above are unclear. However, if we examine the data for the largest and most problematic of the modules (Table 6) we see that authoring of the test cases found none of the major errors while four of the six errors were found during running of the tests. Examination of the errors shows that they all seem to be structural rather than functional in nature.

The total person hours devoted to unit test development and review was such that approximately 6.7% of the total effort was devoted to this activity. A further 3% of the project effort was associated with enabling the unit test to be run in the target environment, a complication due to the small amount of RAM available and the processors segmented architecture.

Table 6 Total (and major errors) in code for the largest module

Activity	Number of errors found
Designing tests	7 (0)
Run on host	6 (4)
Run on target	0

Table 7 Total (and major errors) in code for the largest module

No	Issue/defect	Review	System test
1	Possible infinite loop in EEPROM code	Possible	No
2	Wrong register cleared in eep code	No	No
3	Variable not cleared	Yes	Possible
4	Buffer overflow	Yes	Possible
5	Variable not initialized	Possible	No
6	Off array access (off by one)	Possible	No
7	Arithmetic overflow	Possible	No
8	Timer overflow	Possible	No
9	Numerical accuracy	No	No
10	Memcpy flaw	Possible	No
11	Local variable assigned but not used	Yes	No
12	Redundant logical expression	Possible	No

As part of the project shutdown process, management requested an analysis of the errors found during unit test. In particular they were interested in knowing if the errors located during unit test could have been detected by other means. Table 7 lists the 12 defects that were analyzed as part of this process. Detection is measured against the two other major verification and validation techniques employed: review and bench/system functional test. For each defect examined the analysis gives one of three detection probabilities as follows:

- *No*: it is not believed that this defect could be expected to have been found in the normal course of events.
- *Possible*: it is possible that the defect could have been found but it is not unreasonable that it was not.
- *Yes*: it is reasonable that the defect could have been expected to have been found.

In the case of review the majority of defects either should have been detected or it was felt that it should have been possible to detect the defect. However it was pointed out in the report to management that most errors were potentially detectable by review. For the two defects with a "no" comment it was felt that the amount of work required would have been unreasonable. For example to detect defect 9 it was thought that desk checking[7] would have been required, duplicating the work done in the unit test phase.

The situation with the system tests is more complex and each of the cases is analyzed separately below.

1. To trigger the fault during system test would require injection of either fault inducing code (Voas and McGraw, 1998) into the code under test or physically disabling the on-chip EEPROM.
2. This could not have been found during system test as it has no visible functional effect.

[7] The process of executing the code by hand with a small number of numerical values as opposed to reviewing. Though the former may be a component of the latter this would not be usually done.

3. This was not visible during functional test as it was masked by another defect.
4. It would have been possible to trigger the fault, however it is not clear that the fault would have been detectable unless an extremely long message was sent.
5. This fault had no viable functional effect apart from the first pass though the code i.e., in the first 10ms.
6. This fault requires a catastrophic failure of the system wiring and was not regarded as a credible scenario when designing system tests—because it's not, the system would be completely non-functional before this could arise.
7. It would be almost impossible to trigger given the hardware test environment, for details of this see Ellims et al. (2005).
8. To trigger this during functional test would require unit test like initialization or several years of running.
9. It would be almost impossible to trigger given the hardware test environment.
10. The `memcpy` function is an assembler routine, this fault could not have been triggered in actual use.
11. This fault had no visible functional effect, however writing the unit tests is also a *review* activity as noted above and this was discovered by trying to force data coverage.
12. This fault had no visable functional effect, again the review effect is evident.

Interestingly it is thought possible that the defects 4, 6, 7, 8 and possibly 9 may have been detectable using the PolySpace[8] tool set given their nature and the fact that they were discovered using boundary value analysis.

6. Comparison of the Projects

First, given the differences in the development processes applied, the cost of performing unit testing is reasonably similar i.e., we have a spread of between 5 and 10 percent of total project effort. It is not certain why this spread exists, however it is probable that the requirements for the unit test designs were at least partly responsible. For example Wallace had a looser unit test specification versus the other two projects, formally requiring only statement and boundary value coverage, but probably achieving good data adequacy coverage. The most expensive (%time) project, Grommet required branch and boundary value coverage, but also effectively required MC/DC like coverage for predicates as well has having well defined data adequacy requirements. The adequacy criteria for Sean falls between the other two projects, its formal adequacy criteria being derived from those developed for Grommet.

It should however be noted that even though, in general, only statement or branch coverage was required by the quality plan, the testing seems to do much better. For example, on Wallace, 31% of the functions unit tested obtained 100% linear code sequence and jumps (LCSAJ) coverage (Natafos, 1988; Woodward et al., 1980) which subsumes branch coverage and 66% obtained 90% coverage. In no cases was coverage lower than 80%. What is interesting is that we were not intending to deliberately do better than 100% branch coverage. Why this should be

[8] The basic properties of the PolySpace tool and associated usage issues are outlined in Nguyen and Ourghanlian (2003).

the case is difficult to pin-point from the data available, however the requirements for input test data require good data coverage; work by Sneed (1986) makes a reasonable hypothesis that good data coverage implies good code coverage with well structured code which seems to be supported by some empirical work on applying design of experiments methods to testing (Dunietz et al., 1997).

Table 8 shows the error data given in Section 5 normalized for the number of executable lines of code in each project and for the number of units in each project.

From this data it is difficult to pick out any strong trends as there are too many factors that influence the number of errors that were discovered, for instance; the size of functions, Wallace and Sean have a larger number of small functions used to control access to hardware than Grommet which only implements higher level logic. In addition, the time scales of the projects are different, in particular Sean ran a large number of activities in parallel which tended to both delay error discovery and compound the effect of rework.

The most interesting figures in Table 8 are the rates of error discovery per function, both Grommet and Sean have very similar figures and both projects performed unit testing as the last verification activity. This is perhaps indicative that the number of errors remaining for both projects were similar when unit testing was performed.

The figures for Wallace are however only a factor of two greater, despite the fact that unit testing was performed as the first test activity and hence will (or at lest should) discover more errors than latter activities. This data can probably be used as a 'rule of thumb' on similar projects for predicting the number of errors that can be expected which can be used as an independent check on the adequacy of the test data.

The obvious question is how does the work presented by the studies summarized in Section 2 compare with the work presented in this paper? As the majority of the work is directed at attempting to answer the question "which fault detection techniques are effective?," the answers however are not clear cut and, in some instances, results contradict previous work.

So, as such, none of the work is directly comparable with the results presented here. However we can obtain a partial answer if we examine the work for possible points of comparison.

Firstly a number of the papers contain information on the number of executable lines of code and of the number of test vectors that were applied. This data is summarized in Table 9 which gives the executable lines of code, the number of test vectors and the ratio between the two values. It can be clearly seen that the three projects discussed here have a significantly lower number of lines of code exercised per vector on average.

Table 8 Normalized average number of defects detected on all three projects

Project	Defects	Designing tests	Run on host	Cumulative
Wallace	per 1000 LOC	11.11	9.17	
	per unit	0.41	0.34	0.75
Grommet	per 1000 LOC	2.34	2.99	
	per unit	0.15	0.19	0.34
Sean	per 1000 LOC	10.28 (0.85)	2.28 (1.4)	
	per unit	0.32 (0.03)	0.07 (0.04)	0.39

Table 9 Number of test cases vs. number of executable code lines

	LOC	Vectors	LOC/vector
Wallace	13500	3405	3.9
Grommet	10700	4613	2.3
Sean	3500	1669	2.1
Laitenberger (1998)	262	25	8
Maximulien (2003)	71000	2490	28.5

Some information about the amount of time spent on test activities is also available from a number of the papers[9] and the data has been summarized in Table 10. What is immediately apparent is that the number of lines of code being covered per hour in previous empirical studies differs from what we have observed in industrial practice by as much as an order of magnitude.

However it should be noted that different studies perform the testing activity using different techniques, for example Laitenberger (1998) and Runeson and Andrews (2003) studied only structural testing, while Myers (1978), Kamsties and Lott (1995) performed both functional (e.g., instructing students to pay attention to boundary conditions) and purely structural unit tests.

The unit testing on the three industrial projects (Wallace, Grommet and Sean) perform functional unit testing combined with structural elements to ensure that code coverage requirements were met. If we assume that this would be "normal" industrial practice, then the combined functional/structural LOC per hour figure for Myers[10] (professional engineers) results in 19.5 LOC/hour which is comparable with the Wallace data. Combining Kamsties and Lott's functional/structural data for the first replication (students) gives 32.7 LOC/hour which is only a factor of two greater.

A comparison of the defect detection rates would be desirable, however this is would not be meaningful given the differences in process. The fact that for two of the projects extensive testing occurred before unit testing was applied, thus removing a significant number of errors which could reasonably be expected to have been located by testing would also bias the comparison.

What this is saying about the empirical studies vs. actual industrial practice is not clear, however some differences are obvious. For example in the majority of the studies examined above, the amount of time allowed for each activity was artificially constrained (Myers, 1978) being the notable exception. In the projects examined here that was not the case, at least not to the same degree. Another issue is the subjects training in the techniques applied, or rather lack thereof. In the empirical studies all subjects received some instruction in the techniques to be applied and in some cases undertook "homework" assignments for those activities. However that does not match the continual write, review, rework cycle applied in the projects described here.

[9] The data from Kamsties and Lott (1995) has been separated out for both replications (R1, R2) and only the data for functional unit test has been shown in Table 10. The data as presented here is not given in the paper and the authors have performed there own analysis from the raw data given in the paper from Tables 7.1 and 7.15.

[10] Data for time is taken from Table V in Myers (1978) and the lines of code from pg. 763, the same analysis is performed for Kamsties and Lott (1995) from the raw data presented in Tables 7.1 and 7.15.

Table 10 Lines of code covered per hours for the projects described in this paper and extracted from the existing literature

	LOC	Hours	LOC/hour
Wallace	13500	865	15.5
Grommet	10700	4440	2.5
Sean	3500	400	9
Myers (1978)	83	1.9	43
Kamsties R1/F (1995)	200	2.6	76
Kamsties R2/F (1995)	200	2.2	90
Laitenberger (1998)	262	2	131
Runeson (2003)	200	1.9	105

7. Discussion

Possibly the most significant difference between the projects is when they chose to perform the unit test activities. Wallace had the luxury of having adequate time and stable requirements when unit testing was applied. Where stable requirements were not available Wallace took the path of least resistance and opted to prototype desired functionality avoiding the overheads of applying the full development process. When requirements derived from prototyping activities were stable, the work was reengineered from the requirements down, incorporating unit testing. Neither of the other two projects had the prototyping option though it is felt, in hindsight, that it should have been pursued on Grommet. Sean never had the time nor the adequate manpower to take this approach.

The most notable difference in the approach from the reported figures reported above is the time expended to locate each change, Wallace reported 3.6 hours per defect, Grommet reported 8.4 days and for Sean it is 13.6 days. There are a number of possible reasons for this,[11] firstly Wallace simply had more defects to find as it was the first testing activity applied. Secondly the unit test phase followed directly from the design phase and thirdly the unit test were specified by the module designer. These last two factors possibly make the process more efficient in the sense that the test design process is possibly more effective as a review activity as the designer tends to use there own mental model of what they intended rather than relying wholly on the written documentation. That is they try and test what they thought they designed rather than what they did design. The test design process in this case then tends to compare the mental model vs. the actual design.

There are other possible factors as well, for example over the five years that Wallace ran, the designers became adept at using the THG tool. On the other two projects which were outsourced, this would not have taken place to the same extent.

The unit test data generation activity was outsourced for Grommet and Sean. What effect that has on the quality of the tests is uncertain. At the current time the main issues with the out-sourcing process would appear to be firstly, there is significant training overhead involved to have the test data generated as required. Secondly a significant overhead is associated with monitoring and reviewing the work.

[11] See also comments in Section 5.2.

This second activity occupied 39% of the total effort associated with performing the unit tests on Sean and located 149 required changes to the unit test cases.

In terms of total cost unit testing appears expensive, but not unduly so.[12] However if the cost in terms of man hours to find errors from the Wallace data is considered representative, then unit testing can be between 2 and 13 times more effective than the other test activities applied. The case is less clear cut for the other two projects, however as noted above, engineers were forced to resort to what amounts to unit testing to make progress on Sean.

It has been noted that the attitude of engineers who have been required to perform unit testing changes over time, in effect what we see is Hamlet's "nose rubbing effect" (Hamlet, 1988). The attitude of engineers who in many cases have not performed unit testing previously is interesting to observe. On Wallace two events stand out, during unit test design one engineer enquired "I can't (unit) test this code—can I redesign it?." The answer was of course yes. Another engineer on the same project, who was actively hostile to the idea of performing unit testing commented when he located a bug ".. that one would have been hard to find (elsewhere)." The changing or changed attitude of one engineer who worked on both Grommet and Sean was noted above, the polite version of the comment boils down to "this code should all have been unit tested before we tried to bench (system) test it." Maximilien and Williams (2003) noted a similar effect stating "there was some resistance from developers at first."

The attitude of the Pi Technology authors is also interesting; Ellims undertook the analysis of the available Wallace data to prove to management that unit testing was a cost effective technique. Bridges however was attempting to find evidence for the converse on Grommet, that unit testing was not effective either in terms of errors found and cost. Analysis of the actual error and cost data vs. the *perceived* effectiveness by project engineers shows that the latter premise could not however be justified. The evidence shows that contrary to the *belief* of the project engineers, a non-trivial numbers of errors were located.

The authors also have to acknowledge that there were significant problems in collection and collation of the data. In particular all data had to be extracted after the fact as adequate provision for near real-time analysis had not been made at the start of any of the three projects. However we seem to be learning: data collection on Sean was simpler than on either the Wallace or Grommet projects.

Of the questions proposed at the end of Section 2, we can provide some partial answers. For the first question the Wallace data shows that unit testing appears to be one of the more efficient of the bug removal strategies applied. But it is also clear that review activities are more effective still. It is also clear that even at the end of the development process (Grommet, Sean) there were still faults remaining, despite the application of reviews at all stages and the application of detailed functional testing. In addition to the in-house testing the Grommet code was in use in a vehicle test fleet for nearly two years and functional testing on Sean was also conducted by a completely independent team (the customer). The scale of functional test detail on

[12] It should be noted that the THG tool was developed as part of another large scale development of an engine control unit in response to the ludicrous amount of effort required to do unit testing on a project previous to that. Therefore development costs of the tool are not a major issue for any of the projects discussed here apart from on going maintenance costs.

Sean was impressive, the 3500 LOC were represented by approximately 2240 lines of detailed specification all of which had specific functional system/bench tests. No one technique on its own appears to be completely effective.

As to the second and third questions on resource use we can give definitive answers (Sections 5 and 6) for these three projects which we believe can be directly applied to other projects of a similar nature. However as the projects examined are all so similar, extrapolating the results given here should be made with some caution.

As to the last question, we believe that we have demonstrated that simple adequacy criteria such as statement coverage should be viewed as being inadequate on their own and need to be combined with a data coverage adequacy criteria of some type. Sneed (1986) pointed out the advantage of doing this in 1986 but the general area seems to have suffered some neglect since then: research efforts mainly being applied to adequacy criteria based on coverage of source code. The notable exceptions of course being mutation testing (DeMillo and Offutt, 1993) and more recent work on applying design of experiments methods to software test Cohen et al. (1997).

8. Conclusions

We have examined three projects where we felt that unit testing was done reasonably well. The three projects have a range of different characteristic e.g., fixed point versus floating point implementations, complete systems (Wallace, Sean) versus a project building separate components of a larger system.

On the basis of the very fact that errors were detected by performing unit testing on all three projects it is difficult to argue that the activity is not necessary and so can be cut from the development process. Unit testing seems to hit the spot that other testing does not reach and replicates results from Dupuy and Leveson (2000). However unit tests are difficult to maintain over the life of a project so some practical strategy is required to integrate their use with the customer's and management's desire to reduce time scales and cost (faster, better, cheaper–pick any two).

The data from the Grommet project also provides a strong indication that statement coverage is, as has long been suspected (Tai, 1980; Hamlet, 1994), indeed a poor measure of test adequacy as nearly 97% statement coverage of the complete code set was obtained during functional module testing and errors were still uncovered though unit testing.

Given the strong emphasis on data-adequate heuristics such as boundary value and domain analysis in the unit test guidelines of all three projects, it is thought probable that a large amount of the test effectiveness during unit test should be attributed to those criteria rather than the code coverage criteria. This view is supported by results reported by Jones and Harrold (2003) on the effects of test set minimization on fault detection effectiveness of test sets.

The results which are detailed here are an attempt to evaluate unit testing in context of a complete project and its development process. Many publications have treated unit testing theoretically, compared it qualitatively with other testing techniques or described novel tools. What we have tried to do in this paper is to put some solid results into the public domain and to replace myth with fact. For example for the three large projects reported here managements perceived costs of unit testing were probably exaggerated relative to benefit gained as compared with other techniques such as expanding the system/bench test program.

We hope that the results reported in this paper will have a number of effects:

- That it will encourage other industrial sources to publish empirical data, particularly on testing activities.
- That it will ignite a debate about the role of unit testing in software development.
- Remind the research community that if we want to make progress on responding to hypotheses then the way to do this is through a consideration of real-life project data.

9. Future Work

It is envisaged that the data available on these three projects will allow a number of other studies to be undertaken.

At the suggestion of one of the reviewers the authors intend to re-examine the data collected to see if there are any obvious differences in the nature of the faults that unit testing finds relative to faults that other test activities locate. There appears to be very little data that addresses this. For example Basili and Selby (1987) compared detection rates and characterization of faults. However, it is not clear from their data whether there were faults that were located by only a single technique. A start on this work has been made with the analysis of faults from the Sean project. Unfortunately repeating this analysis on the other two projects will be a massive task[13] and so results could not be included here.

Another avenue that would be of interest to explore would be to examine how adequate the data coverage of the existing test sets is. Recent theoretical work by Dalal and Mallows (1998) and empirical data from Kuhn et al. (Kuhn and Reilly, 2002; Kuhn et al., 2004) indicate that this would be a useful exercise. Unfortunately there are few tools available to make this type of assessment.

One possibility that was actively pursued on the Grommet project was to assess the test inputs for data adequacy via mutation testing. As Hamlet (1995) has stated "this was originally an attempt to fill the data-coverage gap," unfortunately it did not prove possible to find a suitable mutation tool at the time and so this work was not undertaken.

10. Afterword

It's known that testing can only reveal the presence of bugs, not prove their absence. However, having performed unit testing the Pi Technology authors would like to state that they feel much more confident that the residual defect level in the delivered software for all three projects is at a level consistent with current best

[13] For Wallace alone there are over 900 pages of change requests and review records to be reanalyzed.

practice for the automotive industry. It is difficult to put a price on how much this confidence is worth.

Acknowledgments Firstly every engineer the Pi Technology authors have made to "suffer" in the course of these three projects, thank-you. We still think it was necessary. To Dr Charlie Wartnaby who performed the original analysis on detection probabilities in Table 7 and Peter Ladkin for a late review. Secondly to the reviewers for some rather interesting observations and questions.

Appendix: Unit Test Requirements

The following is a summary of the unit test requirements for Grommet and Sean, extracted from the checklists for the software testing specifications for those projects, with some paraphrasing. Where differences exist between the two projects this is noted in italics. It should also be noted that to minimize the space requirements all supporting text has been removed, the test guidelines for Sean are 17 pages in length and contain numerous examples and references.

Small numbers

1. (*Grommet only*) differences should be 1/10 the minimum resolution as defined in the data dictionary.
2. Integer values should vary by a single bit value.

Condition Evaluation

1. Short circuit evaluation—all parts of a condition are to be evaluated once
2. Single relational operator, Test cases should consist of { (A>B), (A<B), (A==B)} with the caveat that the difference between A and B must be as small as possible.
3. (*Grommet only*) Compound statements
   - Where the two conditions are combined with an OR test cases should consist of the following {(A<=B&&C==D), (A<=B&& C<D), (A<=B&&C>D) & (A>B&&C!=D)}.
   - Where the two conditions are combined with AND Test cases should consist of the following {(A>B&&C==D), (A>B&&C<D), (A>B&&C>D) and (A<=B&&C==D)}.
4. The difference between the operands will be made as 'small as possible' to ensure that any error would more likely result in an incorrect path being taken, making the effect of the error more observable.
5. There are a number of problems that seem to occur in code despite all efforts to avoid them
   - one or more inputs missing from a logical expression
   - incorrect comparison operators (!= instead of ==, = instead of == etc)
   - incorrect Boolean operators applied, (&&, ||, !, & instead of && etc)
   - Boolean parenthesis error (incorrect precedence in complex expressions)
   - incorrect relational operator error (using < instead of <=, or < instead of >, etc)

Loops

1. Are loops with variable number of conditions tested to check the boundary conditions.
2. Do loops with a fixed number of iterations execute the correct number of times.
3. Are there any loops which the test designer thinks it may be possible that they do not terminate.
4. Are there nested loops that need to be checked from the inside out?

Data coverage

1. All inputs are tested at their defined minimum values simultaneously and maximum values simultaneously.
2. Each physical value is tested at its range limits, limits to be justified.
3. All outputs are tested to produce a maximum and a minimum range value.
4. At least one test does not cause clipping to take place.
5. All modal values (i.e., enumeration and state variables) have *all* values tested.
6. All modal variables are tested for out of range values.
7. Data structures are tested exercise the limits of their allowed storage (for example the lower and upper limits of arrays).
8. For boundary values the ranges specified in the design and data dictionary should be used.
9. The values selected should include zero and if a signed data type is used both a positive and negative values.

Accuracy of equations

1. All equations are tested for overflow and loss of resolution by giving combinations of boundary value inputs.

   - Where an equation requires either an intermediate or final result to grow larger in magnitude, such as for addition and multiplication, test cases should be chosen such that both arguments are at a maximum amplitude.
   - Repeat test detailed above and, if appropriate, additional test inputs should be chosen so that the final result is as small as possible.
   - Where an equation requires an intermediate or final result to grow smaller in magnitude, such as for division, make the numerator as small as possible and the denominator as large as possible.

2. Fixed-point arithmetic only. Wherever there is a subtraction (A-B), make sure that the test cases are designed such that both A and B are positive and B is larger than A. This will ensure that the signed arithmetic has been handled correctly in case A and B are implemented as unsigned parameters. This test should be applied to both signed and unsigned parameters.
3. Design test cases such that the result of an equation is zero and also the smallest possible number bigger than zero. The purpose of this is to test for loss of resolution.

Unspecified Behavior

1. All equations involving division should have a divide by zero trap implemented.
2. Make sure that all inputs that are not outputs are not changed by the code.

References

Aitchison K, Goody K (2000, September) Effective bench testing of ECUs. Proceedings of ISATA 2000, Automotive Electronics. Dublin Ireland, pp 165–170

Anon (1992) Software Considerations in Airborne Systems and equipment certification, DO-178B. Washington DC, RTCA

Basili VR, Selby RW (1987) Comparing the effectiveness of software testing strategies. IEEE Trans Softw Eng 12(13):1278–1296

Brookes F (1995) The Mythical Man-month. Addison-Wesley, Reading, MA

Burr K, Young W (1998) Combinatorial test techniques: table-based automation, test generation and code coverage. Proceedings of the International Conference on Software Testing, Analysis, and Review (STAR'98). San Diego, USA, pp 26–28

Cohen DM, Dalal SR, Fredman ML, Patton GC (1997) The AETG system: an approach to testing based on combinatorial design. IEEE Trans Softw Eng 7(23):437–444

Dalal SR, Mallows CL (1998) Factor-covering designs for testing software. Technometrics 3(40):234–243

Davis M, Weyuker EJ (1988) Metric-based test-data adequacy criteria. Comput J 1(13):17–24

DeMillo RA, Offutt AJ (1993) Experimental results from an automatic test case generator. ACM Trans Softw Eng 2(2):109–127

Dunietz IS, Ehrlich WK, Szablak BD, Mallows CL, Iannino A (1997) Applying design of experiments to software testing. Proceedings of the 19th International Conference on Software Engineering (ICSE'97). ACM, Boston, USA, pp 205–215

Dupuy A, Leveson N (2000, October) An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. Proc. Digital Aviation Systems Conference (DASC'00). Philadelphia

Ellims M (2000) Hardware in the loop testing. Proceedings ImechE Symposium IEE Control 2000. Cambridge, England

Ellims M, Jackson K (2000) IS0 9001: Making the right mistakes SAE Technical Paper Series 2000-01-0714. Detroit, USA

Ellims M, Parkins RP (1999) Unit Testing Techniques and Tool Support. SAE Technical Paper Series 1999-01-2842. Detroit, USA

Ellims M, Bridges J, Ince DC (2004, November) Unit testing in practice. 15th IEEE Int'l Symposium on Software Reliability Eng (ISSRE'04). Saint-Malo, France

Ellims M, Warnaby C, Wiseman M (2005) Automated functional testing of a battery control system. (Submitted to 12th International Conference Electronic Systems for Vehicles (Elektonik'05) Baden-Baden 6–7 October 2005.)

Fenton NE, Ohisson N (2000) Quantitative analysis of faults and failures in a complex software system. IEEE Trans Softw Eng 8(26):797–814

Frankl PG, Weyuker EJ (1983) A formal analysis of the fault-detecting ability of testing methods. IEEE Trans Softw Eng 3(19):202–213

Freedman RS (1991) Testability of software components. IEEE Trans Softw Eng 6(17):553–564

Garg P (1994) Investigating coverage-reliability relationship and sensitivity of reliability to errors in the operational profile. Proc. 1st International Conference on Software Testing Reliability and Quality Assurance. New Delhi, pp 21–35

Green TRG (1990) The nature of programming. In: Hoc JM, Green TRG, Samurcay R, Gilmore DJ (eds) Psychology of programming. Academic Press

Hamlet R (1994) Connecting test coverage to software dependability. Proc. 5th International Symposium on Software Reliability Engineering, pp 158–165

Hamlet R (1995). Implementing prototype testing tools. Softw Pract and Exp 4(25):347–371

Ince DC, Shepperd MJ (1993) Metrics their derivation and validation. Oxford University Press, Oxford

Jones JA, Harrold MJ (2003) Test-suite reduction and prioritization for modified condition/decision coverage. IEEE Trans Softw Eng 3(29):195–209

Jorgensen PC (1995) Software testing a craftsman's approach. CRC Press, Boca Raton

Kamsties E, Lott CM (1995) An empirical evaluation of three defect-detection techniques. In: Schafer W, Botella P (eds) Proc Fifth European Software Engineering Conference LNCS 989. Springer-Verlag, Berlin, pp 362–383

Kuhn DR, Reilly MJ (2002) An investigation of the applicability of design of experiments to software testing. Proc. 27thAnnual NASA Goddard/IEE Software Engineering Workshop (SEW-27'02)

Kuhn DR, Wallace DR, Gallo AM (2004) Software fault interactions and implications for software testing. IEEE Trans Softw Eng 6(30):418–421

Laitenberger O (1998) Studying the effects of code inspection and structural testing on software quality. Fraunhofer Institute for Experimental Software Engineering. Technical Report ISERN-98-10

Lyu MR, Huang Z, Sze SKS, Cai X (2003) An empirical study on testing and fault tolerance for software reliability engineering. Proc. 14th International Symposium on Software Reliability Engineering, pp 119–130

Maximilien EM, Williams L (2003) Assessing test-driven development at IBM. Proc. 25th International Conference on Software Engineering, pp 564–569

McCabe TJA (1976) Complexity measure. IEEE Trans Softw Eng 4(2):202–213

McDermid JA, Rook P (1991) Software development process models. In: McDermid JA (ed) Software engineers reference book. Butterworth Heinemann, Oxford

Myers GJ (1978) A controlled experiment in program testing and code walkthroughs/inspections. Commun ACM 9(21):760–768

Natafos SA (1988) Comparison of some structural testing strategies. IEEE Trans Softw Eng 6(14):868–874

Nguyen T, Ourghanlian A (2003) Dependability assessment of safety-critical system software by static analysis methods. Proc. of the 2003 International Conference on Dependable Systems and Networks (DSN'03)

Parrish A, Zweben SH (1991) Analysis and refinement of software test data adequacy properties. IEEE Trans Softw Eng 6(17):565–581

Pressman RG, Ince DC (2000) Software engineering: a practitioner's approach (5th edition). McGraw Hill, New York

Richardson DJ, Thompson MC (1993) An analysis of test data selection criteria using the relay model of fault detection. IEEE Trans Softw Eng 6(19):533–553

Runeson P, Andrews A (2003) Detection or isolation of defects? An experimental comparison of unit testing and code inspection. Proceedings 14th International Symposium on Software Reliability Engineering, pp 3–13

Shooman M (1988) Software Engineering. McGraw Hill, Singapore

Sneed HM (1986) Data coverage testing in program testing. Proceedings Workshop on Software Testing. Banff Canada, pp 34–40

So SS, Cha SD, Shimeall TJ, Kwon YR (2002) An empirical evaluation of six methods to detect faults in software. Softw Test Qual Eng 12:155–171

Sommerville I (2004) Software engineering. Addison Wesley, Reading, MA

Tai KC (1980) Program testing complexity and test criteria, IEEE Trans Softw Eng 6(6):531–538

Tichy WF, Lukowicz P, Prechely L, Heinz EA (1995) Experimental evaluation in computer science: a quantitative study. J Syst Softw 28:9–18

Torkar R, Mankefors S, Hansson K, Jonsson (2003) A. An exploratory study of component reliability using unit testing. Proc 14th International Symposium on Software Reliability Engineering, pp 227–233

Voas JM, McGraw G (1998) Software fault injection: inoculating programs against errors. John Wiley and Sons, New York

Voas JM, Miller KW (1995) Software testability: the new verification. IEEE Softw (17)28:17–28

Wood M, Roper M, Brooks A, Miller J (1997) Comparing and combining software defect detection techniques: a replicated empirical study. Proc. 6th European Software Engineering Conference, pp 262–277

Woodward M, Hedley D, Hennell M (1980). Experience with path analysis and testing of programs. IEEE Trans Softw Eng 6(3):278–286

Zhu HA (1996) Formal analysis of the subsume relation between software test data adequacy criteria. IEEE Trans Softw Eng 22(4):248–255

Zhu H, Hall PAV, May JHR (1997) Software unit test coverage and adequacy. ACM Comput Surv 29(4):366–427

Michael Ellims is currently Chief Engineer for Pi Technology working in embedded vehicle control systems for 14 years. Technical interests include embedded electronic control systems, systems safety and hazard analysis and software testing. He has a M.Sc. (Hons.) in computer science from the University of Canterbury and is a member of the IEEE Computer Society and the ACM.

James Bridges is currently a Chief Engineer for Pi Technology and has worked for over 10 years in the field of automotive safety related embedded software controls. After graduating from Reading University with a B.Sc. in Cybernetics and Control Engineering, James was a research engineer for British Telecom working on speech recognition.

Darrel Ince is a Professor of Computing at the Open University, the largest university in the United Kingdom. He is the author of 24 books and over a hundred papers dealing with topics in software engineering. His current research interests are security requirements and software testing.