SOFTWARE VERIFICATION RESEARCH CENTRE

SCHOOL OF INFORMATION TECHNOLOGY

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 00-24

A Pilot Project on Module Testing for Embedded Software

Jason McDonald [*]
Leesa Murray [†*‡]
Peter Lindsay [‡] Paul Strooper [†‡]

July 2000

Phone: +61 7 3365 1003
Fax: +61 7 3365 1533
http://svrc.it.uq.edu.au

Note: Most SVRC technical reports are available via anonymous ftp, from `svrc.it.uq.edu.au` in the directory `/pub/techreports`. Abstracts and compressed postscript files are available via `http://svrc.it.uq.edu.au`

A Pilot Project on Module Testing for Embedded Software

Jason McDonald [*] Leesa Murray [†*‡] Peter Lindsay [‡] Paul Strooper [†‡]

[*] Foxboro Australia, PO Box 4009, Eight Mile Plains, Qld 4113, Australia.
email: { jasonm, leesam }@foxboro.com.au

[†] School of Computer Science and Electrical Engineering, The University of Queensland, Brisbane, Qld 4072, Australia. email: { leesam, pstroop }@csee.uq.edu.au

[‡] Software Verification Research Centre, The University of Queensland, Brisbane, Qld 4072, Australia. email: pal@svrc.uq.edu.au

Abstract

This paper reports on an industrial pilot project with the aim of introducing systematic, automated module testing for embedded software for distributed control systems. The systems are used in safety-related applications and hence have strong requirements for test coverage, auditability and repeatability; in addition, maintenance issues currently dominate software development. Module-level testing is used here to improve test coverage, controllability and observability. This paper explores issues of isolating modules from the run-time environment, improving integration of testing into the development environment, automating testing, and improving test planning and documentation.

1 Introduction

Software testing is traditionally performed at several levels of granularity, including module, integration, and system testing. While integration and system testing are often performed systematically, module testing is sometimes left as the responsibility of the developer and performed in an ad-hoc fashion. Module testing has the potential to deliver better testing coverage and better identification of the location of errors.

This paper reports on an industrial pilot project with the aim of introducing systematic module testing for embedded software. Embedded software in this context, also known as firmware, consists of computer programs and data that are loaded into a class of non-volatile memory, such as EPROM chips and flash cards. The pilot project was carried out within the firmware development group of Foxboro Australia, and occurred during the first year of a three year research collaboration between the Software Verification Research Centre (SVRC) and Foxboro.

The firmware is used in a range of products and has been developed over an eight year period. This period has seen rapid evolution of the product range, and a significant number of hours of use in customer applications has accumulated. For some of the products, their reliable field performance has justified their use as components in safety-related systems. Going forward, however, it is recognised that the safety-related nature of future applications will require that testing of the firmware be carried out systematically. By systematic testing, we mean that the testing is:

- *planned*: to permit design for testability,
- *documented*: so that the test cases can be understood and the adequacy of the test cases can be evaluated (for example, by external auditors or by measuring coverage of the software tested), and

- *repeatable*: so that the test cases can be re-executed after changes in the software.

The requirement for repeatability is especially important for the firmware group, as maintenance and regression testing takes up a significant portion of their work. In this paper, the term "maintenance" includes enhancements, planned modifications as a result of requirements changes, and defect resolution. The requirement for repeatability suggests the need for automating the testing as much as possible, so that the cost of rerunning the tests during regression testing is as low as possible.

Systematic module testing, as described above, is carried out by Foxboro for most of their non-embedded software modules. Similarly, integration and system testing is carried out systematically. However, the module testing of firmware modules is typically left as the responsibility of the developer of the firmware and is carried out in an unstructured fashion.

One of the main goals of the pilot project was to investigate the feasibility of introducing systematic module testing for the firmware. In Foxboro's context, a module is a collection of functions and/or data structures grouped together by their functionality or purpose. A module is the smallest component of a subsystem that is tested.

Firmware modules are typically harder to test than non-embedded software modules because they have poor *controllability* and *observability*. Controllability is the ease with which test inputs can be supplied to the code under test, and observability is the ease with which test outputs can be observed and checked. Both are important testing considerations, since poor controllability and observability make it difficult and costly to automate the testing.

Due to a lack of controllability and observability, it is hard to thoroughly test an individual module during integration or system testing. For example, while it is hard to obtain 100% statement coverage during integration and system testing, obtaining 100% statement coverage during module testing is often straightforward. Similarly, when a firmware module is installed on the target hardware, it typically has poor controllability and observability. This is because module procedures can typically only be invoked on the target hardware by using specialised tools to communicate with this hardware and there is no simple call-based interface. Such tools are also necessary to observe the result of calls to module procedures, such as return values or a change in state.

As a result of the controllability and observability problems, one of the main challenges in the pilot project was to isolate the firmware module under test and to test it on the development platform. We were able to achieve this for three of the four modules tested. For those modules, we fully automated the testing and used the LDRA Testbed coverage tool [22] to confirm that we achieved 100% statement coverage. Analysis of the code and coverage data showed that maximal branch/decision coverage was also obtained. The fourth module was tested on the target hardware, but we were still able to automate the execution of the testing.

In Section 2 we review the related work. We then provide the context for the pilot project in Section 3, by describing the three year collaborative project that it is part of. Section 4 presents the aims and expected outcomes of the pilot project, the data that we decided to collect during the pilot project, and the selection of the four modules that were tested during the pilot project. Section 5 describes the four modules and the testing of those modules. In Section 6 we discuss the data collected during the testing and the significance of this data. Section 7 concludes the paper and presents our plans for future work.

2 Related Work

The literature on techniques and tools for module and other forms of unit-level testing is extensive. For example, in the literature on object-oriented testing [3], considerable attention has been paid to class testing. While this work constitutes important progress, it provides

little practical guidance beyond what is described in standard texts on software engineering [19, 21] and software testing [2, 16].

Specifically, few actual case studies of industrial module testing have been reported. In this area, Fiedler [6] describes a small case study on testing C++ objects, but provides no information on the testing techniques used. The ACE tool, developed by Murphy et al. [15], supports the testing of Eiffel and C++ classes, and provides strong evidence for the benefits of testing classes before testing the system using the classes. The ClassBench framework [11] is designed for the testing of C++ container classes, and has been used to test the container classes of the Standard Template Library [14]. A recent study [20] on retrospective fault data for 17 Ada modules from an industrial project compared statement coverage, branch coverage, random testing, boundary value testing, and several other testing approaches to discover these faults. Boundary value testing outperformed all the other methods, finding nearly all of the faults in nearly all of the 17 modules. In our pilot project, we have used a mix of type-based, boundary-value and implementation-based test cases.

The modules tested in the pilot project are all implemented in C and a number of test execution tools exist for testing C modules. PGMGEN [9, 10] is a research tool that generates test drivers for C modules from test scripts. Similarly, Cantata [13] is a commercial tool that can be used to facilitate the generation of C test drivers. Due to the specialised nature and the overhead involved in learning these tools, it was decided not to use any of them and to hand-code the test drivers for the pilot project. Since they are more widely available and easier to use, we did evaluate a number of code coverage tools for use in the pilot project.

The literature on the testing of embedded software is not very extensive, especially as far as module testing is concerned. A number of authors have addressed the problem of debugging embedded software [7, 8, 18, 24] and some of the issues there are similar to those encountered during module testing. Oshana [17] and Willey [23] discuss statistical testing for embedded systems, which is similar to the statistical testing of more traditional software systems and typically applies at the system level rather than at the module level. Essebag et al. [5] propose a method for the regression testing of an embedded system based on a model of the system and an analysis of what parts of the model have changed to determine what tests need to be re-executed. Again, this work is mostly applicable at the system, rather than the module, level.

3 Collaborative Research Project

Foxboro produces control and automation systems for critical applications, including railways and electricity distribution. The trend in this area is towards increasingly stringent Verification and Validation (V&V) requirements, imposed by customers and emerging standards for risk and reliability. V&V costs are a large and growing component of overall system development and maintenance costs. This is especially the case for software. Oversights in requirements and late discovery of changing requirements result in costly re-engineering.

The SVRC is collaborating with Foxboro to develop process improvements and automated support, with the aim of increasing the integrity of Foxboro's embedded control software while reducing overall lifecycle costs. The collaboration is funded in part by the Australian Research Council, under its Strategic Partnerships with Industry – Research and Training (SPIRT) scheme. Research staff from the SVRC are working closely with staff from the Foxboro firmware group to develop and implement these process improvements.

The project aims to replace essentially unstructured, manual processes by systematic, repeatable, partially automated ones. More specifically, the project is developing frameworks, processes and tools that integrate configuration management, requirements traceability and automated testing.

The project's main research aims are:

- Integrate requirements traceability into the firmware development and maintenance lifecycle.
- Define a configuration management framework which allows subsystems and other major system components to be placed under configuration control, and which improves the characterisation of subsystem versions and changes to subsystems.
- Integrate systematic, automated module testing into the firmware development and maintenance lifecycle.
- Integrate the above three capabilities to improve change management and regression testing.

The improvements will be implemented incrementally, with each of the main focus areas (configuration management, requirements traceability, automated testing) addressed separately at first. Three pilot projects have been undertaken in the first year — one in each of the focus areas. In the testing pilot project, which is described in this paper, we have focused on systematic, automated module testing for firmware.

The improvements are expected to result in more effective use of existing tools, as well as delivering new reporting and assessment functionality. This in turn is expected to increase overall system and software development productivity, simplify compliance with standards- and project-specific development requirements, and reduce costs associated with change management and rework.

4 Pilot Project

Foxboro produces a range of remote terminal units (RTUs) based on the Intel 80x86 and Motorola 680x0 microprocessor families. The firmware group is responsible for providing the embedded software that runs on these *target platforms*. The primary firmware *development platform* used by Foxboro is Windows NT.

The firmware code base contains a set of subsystems, which may or may not be included in particular firmware products, and a core subsystem which is included in every firmware product.

The pilot project trialed module testing processes and code coverage analysis tools for selected portions of the firmware code base. The goals of the project were to:

- Introduce systematic module testing as a process within the implementation phase of firmware that is under development or maintenance.
- Define a template for module test documentation.
- Achieve structured and documented module testing for the selected modules.
- Evaluate code coverage analysis tools during the testing of the selected modules.

Initially a Module Test Description document template was developed based on the IEEE standards for testing documentation [12], and Foxboro's prescribed standard for software test plans. This template was reviewed and approved by project personnel, Foxboro's firmware development manager, and Foxboro's verification and validation manager.

A number of modules were selected from the firmware code base and tested according to their module test descriptions. The testing performed and the test results and coverage data are detailed below.

4.1 Data collected

A Goals-Questions-Metrics (GQM) [1] exercise was conducted to identify metrics data to be collected during the project. The starting goal for the GQM exercise was:

> *Improve the effectiveness and coverage of module testing within the firmware group.*

Based on the GQM exercise, we decided to collect the following data during the pilot project (organised according to the questions from the GQM exercise):

1. How much module testing is carried out?
   - the number of lines of code in each module
   - the total number of lines of code
   - the statement coverage achieved for each module
   - the branch coverage achieved for each module
2. How much does module testing cost?
   - the time taken to develop test descriptions for each module
   - the time taken to develop the test implementation (drivers, harnesses, stubs) for each module
   - the time taken to execute the test cases for each module
3. How many faults were found?
   - the number of faults found during module testing for each module

Several code coverage tools were investigated, with the initial requirements being compatibility with C/C++ and the ability to collect coverage data on both the development and target platforms. All of the tools that were evaluated would have required a significant effort to get them running on the target platform and we were not convinced of the value of collecting coverage data for module tests on the target platform. In general, it is preferable to execute module tests with the module under test isolated from the rest of its environment to achieve better controllability and observability.

The code coverage tool we selected was LDRA Testbed [22], a coverage analysis tool for C and C++. The coverage data we collected with Testbed was statement and branch/decision coverage (branch/decision coverage determines if all boolean terms within decision conditions are executed with both true and false values). Testbed works by inserting additional code (called *instrumentation*) into the source code of the implementation under test. The instrumentation collects information about the statements and branches executed in the code during each test run and saves it for later analysis and browsing. Note that the instrumented code is different from the actual code, which means that the performance of the software is changed when coverage data is collected. Instrumentation does not modify the software's basic functionality, however.

4.2 Modules tested

For the pilot project, three modules were selected from the firmware core and one module was selected from the subsystem implementing the Distributed Network Protocol (DNP) [4].

The core modules lie at the heart of all firmware products. Although they have been heavily exercised over the entire lifetime of the RTU product line, these modules were chosen because structured and documented testing of the firmware core has the potential to benefit many of Foxboro's firmware products. The three modules selected from the firmware core were:

- Points, which has two functions for adding objects to, and retrieving objects from, the internal RTU database.
- Digital Inputs, which has three functions for creating and managing digital (boolean) input objects.
- Control, which has three functions for managing the request and execution of controls (a control is an order for an internal database object or a real output on an I/O device to change value).

DNP is a widely used communications protocol, the specification of which is controlled by a third party, the DNP Users Group [4]. Unlike most of the core firmware, the DNP implementation is recently written and is only used in a subset of the RTU product line. The module selected from the DNP code was File Transfer, which provides functions for remotely reading, writing, and deleting files on the RTU's flash memory card.

5 Module Testing

We now present the modules tested during the pilot project and the testing of these modules. For each module, we present a brief description of the interface of the module, the test environment used to test the module, the test case selection strategy, the test driver, and the results of the tests. We do this in detail for the Points module, and in abbreviated form for the other three modules.

Although we do not discuss it in this paper, it is important to note that test documentation, in the form of a Module Test Description, was written for all four modules tested.

5.1 Points module

The Points module stores a global list of name-object pairs (called a *name list*), sorted in alphabetical order of the name element. This list is used to ensure that the names of all data objects in the RTU are unique.

5.1.1 Interface description

This module has two functions: `Remember()` and `DoYouRemember()`.

`Remember()` is passed a name (a character string) and an "object" (an untyped pointer) and inserts the pair into the name list in the correct alphabetic position. `Remember()` assumes that memory has been allocated for the new name-object pair and a pointer to this memory is also passed in. If the name-object pair is successfully added to the list, zero is returned; otherwise, a negative error code is returned.

`DoYouRemember()` searches the name list for a specified name. If the name is found in the list, is not null and does not begin with a space, a pointer to the corresponding object from the pair is returned; otherwise, a NULL pointer is returned. `DoYouRemember()` is also passed an "inhibit" flag, which controls whether or not error messages are written to a diagnostic message list when the name is not found.

5.1.2 Test environment

For testing, the Points module was separated from the firmware base and tested in isolation. Header files from the base that declared only types and constants were included in the test

environment, and stubs were provided for all function implementations required by the module under test. Several additional functions were provided to permit the test driver to change stub behaviour during testing and to determine whether stubs had been called and what data was passed to them. The tests were conducted by a single-threaded test driver, rather than the embedded, multi-threaded environment of the target platform (combining the different threads in the firmware takes place during integration testing). The output from the test cases was all written to a single test output file for later evaluation.

For the Points module, the following groups of stub functions were provided:

Memory management stubs: The firmware function `staticMallocZero()` is a wrapper around the C language `malloc()` function. We provided a stub for `staticMallocZero()` that allowed the test driver to instruct it (by calling another function before executing a test case) to succeed or fail. This allowed the test driver to easily exercise the parts of the implementation under test that handle memory allocation failures.

`complain()` stub: The firmware function `complain()` is used to write error messages into message buffers, which are viewable on the target platform with a diagnostic program. For testing, we provided a stub for `complain()` that simply writes the error messages to the test output file.

We also provided a function to allow the test driver to reset the ordered list to be empty. This functionality is not provided by the firmware because the name-object list is created once at boot-time and is not modified thereafter. The addition of this reset function provides greater controllability and significantly simplifies the testing of the Points module.

5.1.3 Test case selection

The test design was based on the types of data involved (type-based tests), boundary value analysis, and implementation-based testing. The parameters of each function and the data structures used by them are considered and tests formulated based on their types and boundary values. Consider the name parameter of both functions. The name is a pointer to a string, which may be up to 25 characters in length. We develop tests considering both of these types (string and pointer). Tests based on the name being a string are:

- name is an empty string,
- name is 1 character in length,
- name is greater than 1 character but less than 25 characters in length,
- name is 25 characters in length, and
- name is 26 characters in length.

Tests based on the name being a pointer are:

- name is a null pointer, and
- name is not a null pointer.

Boundary value tests for the name are the same as the type-based tests.

Implementation-based tests, developed by inspecting the implementation and determining tests to fully exercise the code, are:

- name is an empty string, and

- name begins with a space.

Additional implementation-based tests are performed based on the alphabetic sorting of the name-object pairs in the name list.

This test selection process occurs for all parameters of both functions and the global data structure that they manipulate. The code is examined in depth and tests formulated to ensure that all possible paths through the code are executed, all different return values are returned, all error checks are exercised, and all normal case behaviour is executed.

5.1.4 Test driver

The testing, including execution and result evaluation, is automated by a customised test driver written in C. The driver calls the functions under test and generates output in a text file. It records when a failure occurs and overall testing statistics of the total number of test cases executed and the number of failures. Testing is conducted on lists of different sizes, so we supply the driver with a `load()` function which adds a given number of name-object pairs to the ordered list using calls to `Remember()`. A function `check()` is also provided to evaluate the success of the tests using calls to `DoYouRemember()`.

5.1.5 Test results

When the test driver is run, 27,163 test cases were executed, resulting in six action items being raised against the Points implementation for consideration by Foxboro firmware engineers. There were three major faults:

- `Remember()` adds a name-object pair that has an empty string as the name to the ordered list, but `DoYouRemember()` cannot retrieve an object that has an empty name from the list, effectively meaning the object is lost.
- `Remember()` adds a name-object pair that has a string starting with a space as the name to the ordered list, but `DoYouRemember()` cannot retrieve an object that has a name starting with a space from the list, effectively meaning the object is lost.
- `Remember()` does not check that the name pointer is non-null before dereferencing it and does not have an entry condition forbidding null name pointers. This causes the test driver to crash with an invalid page fault.

Three further action items raised against the Points module related to missing and incorrect comments.

The coverage data from Testbed revealed that our test cases achieved 100% statement coverage for both functions, and 82% branch/decision coverage. The branch/decision coverage is the highest attainable, that is, it is not possible to achieve 100% branch/decision coverage for this module without changing the structure of the code. This is a result of the way in which branch/decision coverage is calculated. For example, consider the following statement,

```
if (p == NULL)
    return 1;
```

This `if` statement has a `return` statement in it, which means that the "branch" from the end of this if-statement to the next statement in the program will never be executed. Testbed discovers this and reports it as a deficiency. In addition, `DoYouRemember()` also has an expression in a guard of an `if` statement that can never be false because of the structure of the code.

5.2 Digital Inputs module

The Digital Inputs module manages digital input point objects (similar functionality is provided in other modules for other point types such as analogue input points). A digital input point contains a boolean quantity, a count of the number of times the state of the point has changed, and two flag words which respectively indicate certain static and dynamic properties of digital input points.

This module has three functions: `NewDigital()` creates and initialises a digital input object, `UpdateDigital()` modifies the state of a digital input object, and `ChangeDigital()` provides similar functionality to `UpdateDigital()`, but with a cut-down interface.

The test environment, test case selection criteria and test driver development methods for the Digital Inputs module were the same as those used for the Points module.

A total of 2,181 tests were executed and resulted in 11 action items being raised against the Digital Inputs module. There were three major faults: `NewDigital()` does not check whether one of its parameters is a null pointer before dereferencing it, `UpdateDigital()` has a fault with processing two of its parameters which can have conflicting values (there was no description of what should happen in this case), and `UpdateDigital()` has an operator precedence error which causes a decision condition to be evaluated erroneously. The last error was discovered by analysing the branch/decision coverage data, which revealed that one of the terms in the decision condition was never evaluated to true. The remaining action items were related to missing and incorrect comments, and unsafe type conversions.

The tests achieved 100% statement coverage and 98% branch/decision coverage. The less than optimal branch/decision coverage was for the same reason as explained for the Points module.

5.3 Control module

The Control module manages the request and execution of controls — orders for an internal point or a real output on an I/O device to change value.

Each firmware subsystem contains a queue of control requests for points owned by that subsystem that are awaiting execution (*dispatch* in Foxboro terminology) and a pointer to the dispatch function that performs execution of controls for that subsystem. Control queues are implemented as circular buffers with start position, end position and size counters, a semaphore controlling mutual exclusion, and counting semaphores indicating whether space is available to queue another control and whether a control is available for dispatch.

The control module provides three functions that manage queuing and dispatching of control requests for each subsystem: `piControlQ()` appends an appropriate control request to the control queue of the subsystem to which the controllable point belongs, `Control()` performs some checks on the control request before queuing it by calling `piControlQ()`, and `piProcessControl()` is passed a subsystem's control queue and dispatches the first control in the queue if the queue is in a safe state (start, end and size counters all agree) or clears the queue if it is not in a safe state.

The test case selection criteria and test driver development method for the Control module were the same as those used for the Points module.

The test environment for the Control module is the same as that of the Points and Digital Inputs modules, with the addition of the following groups of test stubs:

Operating system stubs: These stubs replace the semaphore management, critical section entry/exit, and interrupt disable/enable functions that are provided by the real-time operating system used on the target platforms. As the test drivers are single-threaded, the critical section and interrupt function stubs are empty. The semaphore function

stubs implement simple counting semaphores. The test driver calls additional stub functions to set semaphore counts before each test case, to check semaphore counts after each test case, to determine which semaphore operations were called during a test case, and to control timeout behaviour of semaphore operations.

Subsystem management functions: The firmware provides several functions for searching lists of subsystem structures to find which subsystem owns a particular point and which points are owned by a particular subsystem. For testing, we provide stubs that simply return a value that is set by the test driver at the beginning of each test case.

A total of 501 test cases were executed, leading to nine action items being raised on the Control module. Two major faults were found: `piProcessControl()` does not check whether one of its parameters is a null pointer before dereferencing it, and `piProcessControl()` does not correctly implement mutual exclusion on the control queue. The remaining action items consisted of several missing preconditions, incorrect comments, and redundant statements.

The tests for the Control module achieved 100% statement coverage and 100% branch/decision coverage.

5.4 File Transfer module

The Distributed Network Protocol (DNP) permits (among other things) various file transfer operations to be performed over a network. DNP has two variations, Master and Slave. The File Transfer module implements the file transfer functionality that is applicable to the Slave version of the protocol.

The File Transfer module has three functions: `fileWrite()` creates a file or appends bytes to a file in the RTU's flash memory, `fileRead()` reads the entire contents of a file, and `fileDel()` deletes an entire file.

We decided to test the File Transfer module on the target platform, rather than the development platform, for two main reasons. First of all, some testing was already in place for the File Transfer module. This testing was not documented and performed mostly manually using the Dtalk tool. Dtalk is a DNP Master simulator that runs on the development platform and allows the tester to communicate with the RTU via a serial connection. Dtalk was developed within Foxboro's firmware group and provides a menu-driven, interactive interface. Dtalk returns results and data from operations and tests to the PC screen. The tester also verifies that a file exists and checks its properties by using the RTU diagnostic program. Second, it would have taken a significant effort to perform the testing on the development platform, because this would require the stubbing out of functions from the device manager modules, which are responsible for actually reading data from and writing data to memory.

The test case selection criteria used for the File Transfer module were the same as those used for the other modules tested.

Test execution and result evaluation was partially automated. The Dtalk simulator was modified to include the tests for the File Transfer module. Tests could be executed in any order, at the tester's discretion, controlled by menu selections. The simulator returned the result data from each test to the screen for the tester to verify visually, but also determined if the test succeeded or failed, and displayed a message stating this following the result data. The result data was displayed to help the tester determine why a test failed.

A total of 24 test cases were executed, leading to nine action items. Two major faults were found: `fileRead()` does not read valid zero byte files and returns the "file is busy" error code instead of the "file does not exist" error code when attempting to read a file that does not exist. The remaining action items relate to missing and incorrect comments.

As explained in Section 4.1, a decision was made not to attempt to get Testbed running on the target platform, and as such no coverage data is available for the testing of the File

Transfer Module.

6 Metrics and Discussion

This section presents and analyses the data collected during the pilot project, as defined in Section 4.1.

6.1 How much testing is carried out?

Table 1 shows the size of the modules tested and the coverage achieved. The first two rows represent the number of lines of code with and without comments respectively. The third row is the statement coverage achieved using Testbed, and the fourth row is the branch/decision coverage achieved. N/A represents Not Available.

Metric	Points	Digital Inputs	Control	File Transfer
LOC - commented	85	332	678	208
LOC - executable	52	140	233	72
Statement coverage	100%	100%	100%	N/A
Branch/decision coverage	82%	98%	100%	N/A

Table 1: How much testing is carried out?

The total number of lines of code, including comments, for the two firmware areas examined are:

Firmware core:	12,308
DNP Slave:	31,393

As Table 1 shows, we achieved 100% statement coverage for the three firmware core modules. As explained for the Points and Digital Inputs modules, even though the branch/decision coverage for these modules is not 100%, it is the highest attainable without changing the structure of the code.

The process of trying to achieve 100% branch/decision coverage for the Digital Inputs module revealed a problem with `UpdateDigital()`. Engineers in the firmware team recognised the problem as being associated with a problem that had manifested itself in the field. This observation has demonstrated the positive impact that module testing can make on product reliability in the field.

6.2 How much does module testing cost?

Table 2 shows the time taken to develop the test descriptions and implementations, and the time taken to execute the test cases. For each module, each test case is executed once per test run. Multiple test runs were conducted for each module.

Metric	Points	Digital Inputs	Control	File Transfer
Test description	15 hrs	11.5 hrs	2.5 hrs	24 hrs
Test implementation	73 hrs	56 hrs	24 hrs	80 hrs
Test execution	10 sec	$<$ 1 sec	10 sec	3 min

Table 2: How much does module testing cost?

The cost of formulating test descriptions is low considering the benefits they provide. The test descriptions provide a documented record of the testing conducted, including test cases and descriptions of the test environments developed and used. This allows the testing to be repeated by other firmware and test engineers. Results of the testing conducted are kept as attachments to the test descriptions.

The cost of developing test implementations is high, especially considering the lines of executable code tested (286 hours for 497 lines of code). However, the cost is not as high as the data in Table 2 indicates if we analyse further what was achieved. The Points module was the first firmware module tested, and the testing was conducted by SVRC personnel — not firmware experts. Initially it was tested by removing the two functions to a file of their own. Then this testing was repeated, leaving the functions in their original file and writing test harnesses and stubs, as explained in Section 4, to allow this. The time also includes familiarisation with Testbed, getting it to run with our compiler, and then using it on the Points module. The time taken to achieve all of these tasks makes up the 73 hours logged against the Points module's test implementation in Table 2.

The Digital Inputs module was tested next. It was also tested by SVRC personnel. The lessons learnt from the Points module testing meant that developing the test harnesses and stubs was much easier. However, the functionality of Digital Inputs is much more complex than that of the Points module, and a lot more time was spent developing test cases to ensure thorough testing. Achieving maximal code coverage was also much more challenging due to the complexity of the code.

The last core module tested, Control, has a relatively low cost for test implementation development. This testing was conducted by a Foxboro verification and validation engineer. By the time Control was tested, we were much more familiar with the firmware and the code coverage tool, which is reflected in the data in Table 2.

The File Transfer module's testing was conducted by a Foxboro firmware engineer, who we consider to have expert knowledge of the module and its functionality. SVRC personnel assisted with test case selection, witnessed the final testing of the module, and recorded the results. The major achievement for this module was the automation of the testing by enhancing an existing protocol simulator, as explained in Section 4. We consider this a major achievement as the testing was conducted on the target platform, but we still managed to achieve good controllability and observability of the code under test. The automation of the File Transfer module's testing accounts for the majority of the time attributed to its test implementation in Table 2.

What the firmware module testing exercise has revealed is that to reduce the costs involved and make future module testing easier, we need to develop a generic firmware testing environment for the development platform. The firmware code base is currently set up, through compiler switches, to compile for two target platforms. We could add a third compilation option to set up a testing environment on the development platform. This would greatly reduce the cost of future testing of firmware core modules.

6.3 How many faults were found?

In relation to the faults found during module testing, it is important to recognise that faults in a module do not always translate to failures in the field. That is, a module fault will only manifest as a failure in the field if the use of the module in its target environment exercises the logical path that contains the fault.

It should also be noted that some of the faults that were found during module testing would most likely also have been found during a code review. Most minor faults below fall into this category. The relation between code reviews and module testing, such as costs involved and the types and number of faults found by each of these activities, is an interesting issue that deserves further attention.

Table 3 shows the faults found during the testing, classified as either major or minor. A major fault is one that caused a test case to fail or the program to crash. Minor faults include missing or incorrect comments, missing preconditions, and unsafe type conversions.

Metric	Points	Digital Inputs	Control	File Transfer
Major faults	3	2	2	3
Minor faults	3	9	7	6

Table 3: How many faults were found?

We believe the results from the pilot project show that module testing is feasible for firmware. One major benefit of this module testing is that regression testing of these modules can now be conducted at a very low cost. This is due to the testing being automated through the test drivers we have developed, and the testing being documented in the test descriptions.

Another benefit from the module testing is the revelation of the faults, which are discussed in Section 4 and summarised in Table 3. The firmware core code has been used in many applications, and has accumulated substantial hours of use in the field. Despite such a reliable track record, our testing was able to reveal faults. Rectification of these defects will improve the reliability of future applications with operational profiles that would have exposed such faults. When compared to the current practice of testing firmware, our module testing allowed us greater observability and controllability, making it possible to reveal these faults.

The achievement of statement and branch/decision coverage data for the firmware core modules is another significant benefit. This exercise demonstrated that once we were familiar with Testbed, it was relatively straightforward to collect this coverage data, on the development platform. During this pilot, we decided not to try to get the code coverage tool running on the target platform, and this provides us with a challenge for the future.

7 Conclusions

We have described a pilot project that applied systematic module testing to modules developed by the firmware development group of Foxboro. By systematic testing, we mean that the testing is planned, documented, and repeatable. We have achieved the pilot project goals of defining a template for module test documentation, performing systematic module testing for the four modules investigated, and using a coverage tool for three of the four modules.

The results of the pilot project show that module testing is feasible and the testing of the four modules included in the pilot project revealed a number of problems. For three of the modules, we were able to isolate the module under test and carry out the testing on the development platform. Module isolation was achieved mostly by writing stubs to provide adequate controllability and observability. For these three modules, we used the Testbed coverage tool to measure coverage and achieved 100% statement coverage. The fourth module was tested on the target hardware platform, which meant that we did not use a coverage tool, but we were able to automate the execution of the testing.

The cost of setting up the module testing for these four modules was non-trivial. However, a significant portion of this cost was associated with isolating the modules from the run-time environment, which was set up for the real-time operating system used on the target platform. This cost could be significantly reduced by setting up a similar run-time environment to permit testing on the development platform.

The pilot project was part of the first year of a three year collaborative project between the SVRC and Foxboro. In the next two years, we plan to build on the results of this and two other pilot projects in the areas of configuration management and requirements traceability.

In particular, we will implement the recommendations from each of the pilot projects. For example, we will apply systematic module testing as described in this paper to the modules of a control timetable subsystem in order to demonstrate a high degree of code coverage. This module testing will be compared against the manual testing that is currently carried out for the same modules. One of the reasons this subsystem is selected is that some requirements changes are anticipated in the near future, and we hope to be able to reuse much of the automated module testing for regression testing after the changes.

The other main challenge in the overall project will be to integrate the results of the pilot projects into a coherent framework to improve change management and regression testing.

Acknowledgements

The work reported in this paper is supported by the ARC Strategic Partnerships with Industry - Research and Training (SPIRT) grant C49937058. We thank the firmware group at Foxboro for their assistance in this pilot project. In particular, we thank Paul Ellis and Alena Griffiths for their work as part of the steering committee, and Tom Chiu for his efforts in testing the File Transfer module. We also thank Brenton Atchison, Paul Ellis, Alena Griffiths, and Anthony MacDonald for comments on earlier drafts of this paper.

References

[1] V.R. Basili and H.D. Rombach. The TAME project: towards improvement-oriented software environments. *IEEE Transactions on Software Engineering*, 14(6):759–773, 1988.

[2] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, 2nd edition, 1990.

[3] R. V. Binder. Testing object-oriented software: a survey. *Software Testing, Verification and Reliability*, 6:125–252, 1996.

[4] DNP Users Group. *DNP V3.00 Subset Definitions*, 1995. Document number: P009-01G.SUB, Version 1.0.

[5] S.H. Essebag, S-T. Levi, Y. Shai, and N. Bechor. Applying a contamination model to testing an embedded system. In *Embedded Systems Conference*, 1998. http://www.embedded.com/.

[6] S.P. Fiedler. Object-oriented unit testing. *Hewlett-Packard Journal*, pages 69–74, April 1989.

[7] J.G. Ganssle. Debuggable designs. *Embedded Systems Programming*, 11(13):97–100, 1998.

[8] C.A. Haller. On-chip-debug simplifies embedded-system test. *Test & Measurement World*, 17(7):65–70, 1997.

[9] D.M. Hoffman. A CASE study in module testing. In *Proc. Conf. Software Maintenance*, pages 100–105. IEEE Computer Society, October 1989.

[10] D.M. Hoffman and P.A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, 1995.

[11] D.M. Hoffman and P.A. Strooper. ClassBench: A methodology and framework for automated class testing. *Software: Practice and Experience*, 27(5):573–597, 1997.

[12] IEEE Computer Society. *IEEE Standard for Software Test Documentation*, 1983. ANSI/IEEE Std. 829–1983.

[13] Cantata for C. http://www.iplbath.com/p4.htm.

[14] J. McDonald, D.M. Hoffman, and P.A. Strooper. Programmatic testing of the Standard Template Library container classes. In *Proceedings of IEEE Intl. Conf. Automated Software Engineering*, pages 147–156, November 1998.

[15] G. Murphy, P. Townsend, and P.S. Wong. Experiences with cluster and class testing. *Communications of the ACM*, 37(9):39–47, 1994.

[16] G.J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[17] R. Oshana. Statistical testing techniques for embedded systems. In *Embedded Systems Conference*, 1999. http://www.embedded.com/.

[18] K.H. Peters. Software development and debug for system-on-a-chip. In *Embedded Systems Conference*, 1999. http://www.embedded.com/.

[19] R.S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, fifth edition, 2000.

[20] S.C. Reid. Module testing techniques–which are the most effective? In *Proc. of Eurostar97: The Fifth European Conference on Software Testing*, November 1997.

[21] I. Sommerville. *Software Engineering.* Addison-Wesley, fifth edition, 1996.

[22] Overview: What is LDRA Testbed, and how can it help you? http://www.ldra.co.uk/1999/core-info/overview.htm.

[23] H.M. Willey. Engineering specifications: The road map of reliability testing. In *Embedded Systems Conference*, 1999. http://www.embedded.com/.

[24] D.J. Wisehart. Debugging embedded systems. *C/C++ Users Journal*, 17(6):24–34, 1999.