CASE STUDY OF 100% TEST COVERAGE

J.C. SIMNER*, J. CONWAY*, T. OSMAN** and D. AL-DABASS**

** Siemens Communications*
Technology Drive, Beeston,
Nottingham, NG9 1LA.
john.simner@siemens.com

*** School of Computing & Mathematics*
The Nottingham Trent University
Nottingham, NG1 4BU.
taha.osman@ntu.ac.uk

Abstract: Telecommunication software is expected to have a long lifespan during which many developers will add new features, modify existing features, or correct bugs. The software must be understandable, reliable, and maintainable otherwise the additions and modifications will take longer to develop and introduce further errors. Siemens has a software development process, which includes Fagan inspections, module testing, integration testing, internal, and customer field trials. The Quality Plan for a large software development stated that the developed software will be subject to either "100% code reviews and normal levels of testing" or "No code reviews and 100% testing". This paper tries to determine whether "100% testing with no code reviews" is a viable alternative to "100% code reviews with testing" for commercial software products. It provides a case study of a recently developed application that was subject to "100% testing and no code reviews". It uses a commercial tool to demonstrate the test coverage.

Keywords: quality assurance, telecom software testing.

1. INTRODUCTION

Telecommunications plays an important role in any company's business. Modern private telecommunication equipment provides an extensive range of features which allows companies to manage their business more efficiently and effectively. As their business, organisation, or needs evolve, it is imperative that the configuration of the installed equipment and network is changed to meet the new circumstances otherwise, the efficiency and effectiveness of their business may be less than it should be.

Over the last two years, Siemens Information and Communication Networks have developed an enhanced Administration and Service (A&S) product which allows easier access to the configuration and performance data on Siemen's range of telecommunication equipment. Figure 1 shows an overview of the A&S product (known as "HiPath 4000 Manager").

The HiPath 4000 Manager was developed by a multi-national and multi-site team, involving 800 people worldwide, including Beeston, Munich, Berlin, Graz, and Boca Ratoon.

The goal of this paper is to provide a case study of a recently developed application that was subject to 100% testing and no code reviews.

One of the applications developed for the HiPath 4000 Manager (PmAmoProc) was chosen to be subject to 100% testing and no code reviews. It is a standalone application that runs on the HiPath 4000. It periodically invokes collection commands, parses their output, and generates statistic reports, which are collected and handled by two other applications on the HiPath 4000 Manager.

Section 2 reviews software inspection and testing techniques to determine whether "100% testing with no code reviews" or "100% code reviews with testing" are viable alternatives.

Section 3 identifies a testing strategy that should achieve 100% test coverage using both black-box and white-box testing approaches.

Section 4 uses the same to dynamically analyse the product and determine what percentage of the product was actually tested. Initially, it identified a high percentage of untested paths. The reasons for the untested paths were determined. Additional tests were carried out until there was no further increase in test coverage. 100% test coverage was not achieved because some exception handling code could not be executed. As this situation is not unique to this particular product, an additional utility is available to mark the exception branches. This effectively removes them from the metrics thereby achieving higher overall test coverage.

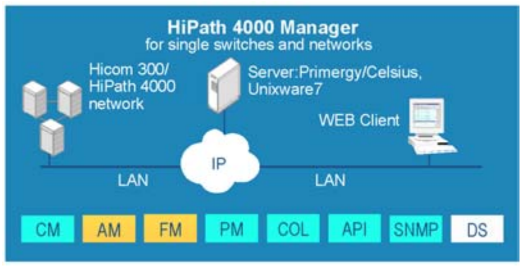

HiPath 4000 Manager
for single switches and networks
Hicom 300/ HiPath 4000 network
Server:Primergy/Celsius, Unixware7
WEB Client
IP
LAN
LAN
CM
AM
FM
PM
COL
API
SNMP
DS

Figure 1 – HiPath 4000 Manager Product Overview

Section 4 concludes with a fault analysis of the errors found during testing and field trials to determine whether the decision made to perform no code reviews was correct or not.

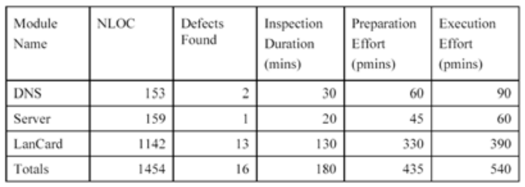
Section 5 summarises how well the chosen application met its goal and demonstrated reliability and maintainability. It identifies that no code reviews and 100% testing leads to coding errors being detected in field trials that should have been found earlier.

2. PRODUCT QUALITY

Telecommunication software is expected to have a long lifespan during which many developers will add new features, modify existing features, or correct bugs. The software must be understandable, reliable, and maintainable otherwise the additions and modifications will take longer to develop and introduce further errors.

Siemens has a software development process, which includes Fagan inspections, module testing, integration testing, internal, and customer field trials. The Quality Plan for the HiPath 4000 Manager states that the developed software will be subject to either "100% code reviews and normal levels of testing" or "100% testing and no code reviews". This section explores how realistic this is.

Table 1 shows the actual code review statistics for three modules in the HiPath 4000 Manager. In summary 1454 lines of code were reviewed in 3 hours consuming 16¼ person hours of effort and finding 16 defects.



Module Name	NLOC	Defects Found	Inspection Duration (mins)	Preparation Effort (pmins)	Execution Effort (pmins)
DNS	153	2	30	60	90
Server	159	1	20	45	60
LanCard	1142	13	130	330	390
Totals	1454	16	180	435	540

Table 1 - Code Review Statistics

Laitenberger and DeBaud (1998) carried out a survey of Software Inspection Technologies. He cites the work of Ackerman et al. (see Laitenberger and DeBaud 1998, p19) which reports the individual preparation and meeting time, per thousand lines of code, for code reviews, by two different development groups; 7.9 and 4.4, and 4.91 and 3.32.

The figures for HiPath 4000 Manager are 1.66 and 2.62. This shows that the review rate on HiPath 4000 Manager compares favourable with other organisations whilst the preparation rate is substantially higher. The difference may be due to knowledge, familiarity, or complexity of the code.

In the HiPath 4000 Manager, there are approximately 14000 modules with 3.5 million lines of code. At the above review rate, it would take 20 person years of effort to review all the code (assuming an 8 hour day and 20 working days per month).

Laitenberger and DeBaud report that "part of the problem [with software inspections] is the perception that … [they] cost more than they are worth." (Laitenberger and DeBaud 1998, p21).

For example, is it worth spending 20 person years (@ £85K per person per year, total £1.7 million) inspecting 3.5 million lines of code. Expressed in this way the general answer is no. It is not economical or viable to do so.

So, is 100% testing more realistic?

Rushby (1991), Watson (1996), and Watson and McCabe (1996) all describe different testing methodologies. They include; random, regression, thorough, and functional.

Watson and McCabe identify that "[a] common approach to testing is based on requirements analysis. A requirements specification is converted into test cases, which are then executed … " (Watson and McCabe 1996, p2).

This is a very easy approach to adopt. If the software has been analysed and designed to meet the requirements then executing the test cases will fully test the software. Watson and McCabe identify that the requirements are usually at a higher level than the code so a lot of the code will not be tested.

Therefore, a lower level approach must be adopted. The code is inspected and a set of test cases is derived that test each and every statement, line, branch, variable, or path, through the code. This is a manual method, which is very time consuming and error prone.

Watson outlines the different testing criterions including; statement testing, code coverage, branch testing, data flow testing, and structured testing.

Watson states that "Structured testing, also known as basic path testing, is a methodology for software module testing based on the cyclomatic complexity measure of McCabe." (Watson 1996, piii and p1).

The cyclomatic complexity measures the logical complexity of a module.

Watson and McCabe state that "it gives the number of recommended tests for software." (Watson and McCabe 1996, p7). A module's cyclomatic complexity is the minimum number of tests required to fully test the module. It is a theoretical value, which may not be achievable in practice.

Watson and Watson and McCabe both define and characterise cyclomatic complexity.

Watson and McCabe cite the work of McCabe stating that "Structured testing is more theoretically rigorous and more effective at detecting errors in practice than other common test coverage criteria such as statement and branch coverage." (Watson and McCabe 1996, p31).

Watson identifies an automated approach to structured testing in which the source code is instrumented and writes a trace file of its execution. The McCabe INTEGRATED QUALITY™ toolset is a commercial tool that automatically instruments the source code and analyses the resultant trace file. The tool reports code, branch, and complexity coverage.

There are very few papers on structured testing and the McCabe toolset. There are many papers on metrics, some of which question the usefulness and theoretical foundations of the cyclomatic complexity metric. In their defence, Watson and McCabe present many case studies that report successes with cyclomatic complexity.

Finally, there is general concensus that 100% testing is feasible using an automated tool to record test coverage. However, it is still a high risk strategy to perform 100% testing in preference to code inspections especially as Laitenberger and DeBaud reports that "available quantitative evidence [between 19-93% of all defects were detected by inspections] … indicates that inspections have had significant positive impact on the quality of the developed software and that inspections are more cost-effective than other defect detection activities, such as testing." (Laitenberger and DeBaud 1998, p21).

3. TESTING STRATEGY

The case study adopted a two stage testing strategy. A black-box testing approach was carried out in the first stage with a white-box testing approach in the second stage. These two approaches are not alternatives; they are complimentary. They are normally applied at different stages in the development of the code. This particular testing strategy was adopted to see how much test coverage was achieved by each approach.

Black-box testing tests the functionality of the software at its interfaces. The tests are usually performed at the end of the coding stage. They check that the software meets its requirements. White-box testing (which includes basic path and control structure testing techniques) tests the code from a design perspective.

With knowledge of the code and its data structures, a set of test cases can be derived that test the individual paths, controls, and data within the code. These tests can be performed whilst the code is being developed. Experience has shown that most errors are logical ones. They occur through incorrect equality tests (e.g. a < b, a <= b, etc.) and incorrect boundary conditions on loops. These errors are easily detected by white-box testing.

With white-box testing, 100% test coverage should be achieved but it takes a long time. Whereas, with black-box testing, all of the functionality can be tested in a reasonable time frame but 100% test coverage may not be achieved. The correct testing strategy will use the best combination of both approaches to find the maximum number of errors in the minimum time whilst ensuring maximum test coverage.

The individual test cases are described in Appendix A.

MCCABE DYNAMIC ANALYSIS OF PMAMOPROC APPLICATION

A commercial tool, McCabe INTEGRATED QUALITY™ Test tool was used to measure and assess the effectiveness of the testing strategy. The tool automatically instruments the source code so that it can identify what has been covered. It supports a number of different module testing methodologies; structured testing, design path coverage, branch coverage, slice coverage, and Boolean coverage (McCabe 2001, p116).

Path, code, and branch coverage are all used to assess how well the PmAmoProc application is tested using the test cases defined in appendix A.

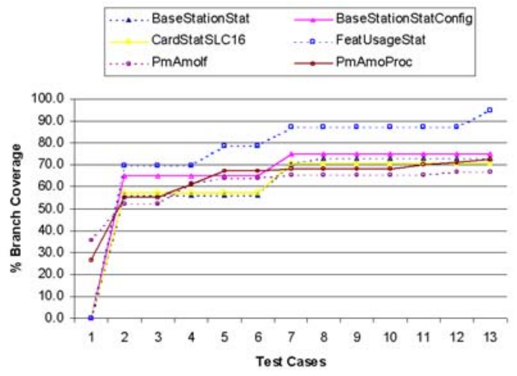


Figure 2 - Class Coverage Metrics

Figure 2 test cases 1 and 2 show the branch coverage for the six classes in PmAmoProc. It clearly shows that only 52.4% to 69.6% branch coverage has been achieved by black-box testing. This confirms the Watson and McCabe (1996) finding that test cases based only on requirements do not test most of the code.

The McCabe INTEGRATED QUALITY™ Test tool was used to identify untested paths, determine why they were not tested, and derive suitable test cases.
The reasons for the untested paths were as follows:

1. **High Level Requirements** - Watson and McCabe identified that requirements may be at higher level than the code so code inspection may be required to generate test cases. This approach was used to generate the latter test cases listed in appendix A.

2. **Boundary Checks** – All parameters received from the command line or external modules should be range checked/validated. With embedded software and no debugging environment, it may be difficult to generate out of range values from external modules.

3. **Test Code** – There are many different approaches to testing. One approach is to write all the code then test it whilst an alternative approach is to write some of the code, test it, then write some more. To perform incremental testing, test stubs or test options are added but rarely removed. The additional code, tested or not, affects the test coverage metrics.

4. **Trace Code** – It can be very difficult to debug embedded software on target hardware so trace statements are often added. As they always effect performance, they must be enabled or disabled through compiler options or control files on the target hardware. A control file is the preferred option because it allows tracing to be turned on and off in the field whilst the compiler option requires a new version to be built, distributed and installed. The software is often tested with tracing enabled. It must also be tested with tracing disabled to ensure maximum test coverage.

5. **Unreachable Code** – Sometimes modules (or code) are written that can never be reached. For example, a class must always have a default constructor. If the class also has a non-default constructor, the default constructor may never be called. The additional code effects the test coverage metrics.

6. **Error/Exception Handling** – Embedded software must detect and handle all error conditions (or exceptions) to ensure that the software continues to run and does not cause any problems. For example, if the application writes to a file, it must always check that the file has been successfully opened before it writes to it. If it fails to open the file, it should log an error rather than write to it. It is very difficult to test system errors and exceptions.

7. **Semaphore/Control Files** – With multi-threaded and multi-process applications, there is controlled access to shared resources. With embedded software and no debugging environment, this may be difficult to test.

8. **Redundant Code** – Sometimes modules are written but never called. The additional code effects the test coverage metrics. They should always be removed.

9. **Coding Error** – Sometimes coding errors occur that results in unreachable code, which is not detected by the compiler. For example, in PmAmoProc, a class instance is explicitly created but never deleted. This coding error should have been picked up at code review. It effects the test coverage metrics. It may cause memory leaks at run time.
The derived test cases and their class coverage metrics are listed in appendix A. Figure 2 shows a graph of branch coverage versus the test cases for the six classes in PmAmoProc. Branch coverage was chosen because it gave the highest coverage value. However, Figure 2 clearly shows that only 66.9% to 94.9% branch coverage has been achieved with the derived test cases. The discrepancy is due to boundary checks, unreachable code, and error/exception handling.

Table 2 lists the path, code, and branch coverage for the least tested modules after completing all of the test cases listed in appendix A.

Module Name	% Coverage			
	v(G)	iv(G)	Lines	Branches
BaseStationStatConfig::~BaseStationStatConfig	0	0	0	0
PmAmoIf::checkAmoSuccessfullyCompleted	25	28.6	34.6	46.7
PmAmoIf::convertAndStripSpaces	25	33.3	100	57.1
PmAmoProc::refreshSLC16details	18.2	18.2	100	61.9
PmAmoIf::setupMpcidInterfaceDirect	33.3	33.3	29.7	63.6
PmAmoIf::setupFamosSession	22.2	25	66.7	64.7
PmAmoIf::getBaseStationStatFromRegZielOutput	0	0	83.9	69.6
PmAmoIf::getSLC16CardStatFromRegZielOutput	0	0	89.5	72.7
PmAmoIf::getDateTimeFeatStatsFromDisZauslOutput	0	0	85.7	75
PmAmoIf::getSLC16CardPositionsFromDisBcsuOutput	0	0	90.9	90.9
PmAmoIf::getStationNumbersFromDisSdsuOutput	0	0	93.6	91.3

Table 2 - McCabe Dynamic Analysis – Least Tested Modules

A module's cyclomatic complexity (v(G)) is the minimum number of tests required to fully test the module. It is a theoretical value, which may not be achievable in practice (see McCabe 2001, p51-53).

A module's design complexity (iv(G)) is the number of paths with calls to other modules (see McCabe 2001, p55-57). If there are no calls to other modules, the design complexity is zero and can not be tested.

Line coverage shows how many source code statements were executed. This does not include any comments or blank lines, as they can never be tested.

Branch coverage shows how many of the exits from branches were executed. For example, an ‘if … then’ has two exits, one if the condition is true and one if the condition is false. A ‘if … then … else’ also has the same two exits.

An analysis of the modules listed in Table 2 revealed:

- **1 Coding Error –** For example, an instance of BaseStationStatConfig was created by PmAmoProc and never deleted.
- **4 Boundary Checks –** For example, checkAmoSuccessfullyCompleted() checked the AMO output for NOT COMPLETED, NOT EXECUTED, and EXECUTED. These conditions were not generated during the tests.
- **31 Error/Exception Handling –** For example, convertAndStripSpaces() checked for null strings and strings with no spaces. These conditions were not generated during the tests.

This analysis clearly shows that it is extremely difficult to achieve 100% test coverage because of the difficulties testing exception handling code.

As this situation is not unique to these particular modules, McCabe have developed an additional utility (McCabe Exception Coverage Utility Version 1.8 20020228) that can mark the exception branches. The branch report shows the percentage of branches tested and the percentage of non-exceptional branches tested (i.e. the number of actual branches minus those marked as exceptional). This effectively removes them from the metrics thereby achieving higher overall test coverage.

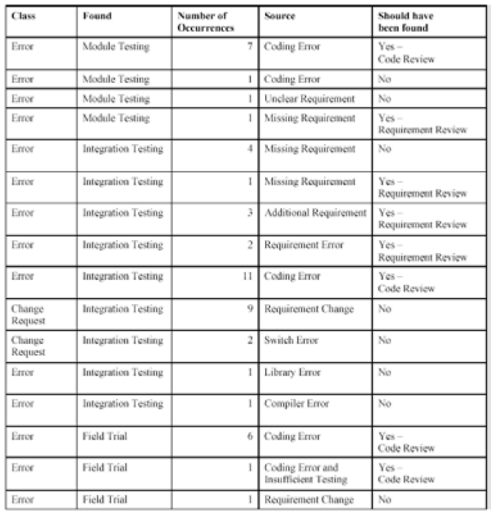
Finally, Table 2 shows 5 modules with low cyclomatic and design complexity coverage but high code and branch coverage. The untested graph listings in the McCabe INTEGRATED QUALITY™ Test tool did not match the cyclomatic complexity coverage. This matter was raised with McCabe. The calculation of the cyclomatic and design complexity coverage in the test tool is designed for modules with only one entry and exit point. If the code allows early exits from the module by exiting straight out of a loop, the coverage for that module may not be recorded. There is some debate over whether it is good or bad practice to exit straight out of loops. The author believes it is sometimes permissible if it makes the overall code simpler.

4. RESULTS AND DISCUSSION

The main intention of this case study was to subject a developed application to 100% testing and not do code reviews. Section 0 identified that this is a high-risk strategy, which may not find all of the code errors. One way of determining that the strategy has worked is to carry out a fault analysis at the end of the field trial. The fault analysis determines; how many errors were detected, where they were found, their cause, and whether they should have been found earlier. The fault analysis was extended to cover the whole development cycle (i.e. development phase, module testing, integration testing, and field trials).

If the strategy has worked, none of the errors raised during the field trial will have a cause of coding error.

Table 3 shows a summary of the fault analysis for PmAmoProc.



Class	Found	Number of Occurrences	Source	Should have been found
Error	Module Testing	7	Coding Error	Yes – Code Review
Error	Module Testing	1	Coding Error	No
Error	Module Testing	1	Unclear Requirement	No
Error	Module Testing	1	Missing Requirement	Yes – Requirement Review
Error	Integration Testing	4	Missing Requirement	No
Error	Integration Testing	1	Missing Requirement	Yes – Requirement Review
Error	Integration Testing	3	Additional Requirement	Yes – Requirement Review
Error	Integration Testing	2	Requirement Error	Yes – Requirement Review
Error	Integration Testing	11	Coding Error	Yes – Code Review
Change Request	Integration Testing	9	Requirement Change	No
Change Request	Integration Testing	2	Switch Error	No
Error	Integration Testing	1	Library Error	No
Error	Integration Testing	1	Compiler Error	No
Error	Field Trial	6	Coding Error	Yes – Code Review
Error	Field Trial	1	Coding Error and Insufficient Testing	Yes – Code Review
Error	Field Trial	1	Requirement Change	No

Table 3 - Summary Fault Analysis for PmAmoProc

The fault analysis showed that 41 faults and 11 change requests were found on the PmAmoProc application.

The faults were categorised as:

- Unclear, Missing, Changed, and Additional Requirements – 22
- Coding Errors and Insufficient Testing – 26
- External Errors (i.e. Switch, Library, and Compiler) - 4

This clearly demonstrates one of the problems of developing software is the nature of the requirements. They are often unclear and change during the development.

The coding errors were examined to determine the cause of the high fault rate (21 faults/KLOC).

There were three classes, which were very similar in functionality. The code from one class was copied to the other classes and amended as appropriate. There were coding errors in the first class, which were also copied into the other classes; resulting in a higher fault rate.

The remaining coding errors were due to over zealous error reporting.

Embedded software must detect and handle all error conditions (or exceptions) to ensure that the software continues to run and does not cause any problems.

The PmAmoProc application is a good example. It is expected to automatically run on a periodic basis, collect the information, and produce the reports. There is no user intervention. If it fails to run, collect the information, or generate the reports, the historical information for that period will be lost forever.

There are two ways of developing the error or exception handling code; pre-emption or responding to crashes.

The first approach looks at the overall system, tries to determine what might cause errors, and adds exception code to handle these situations. The exception code usually reports an error and recovers from the situation. There are two potential problems with this approach; it may identify errors that are not errors, and it will never find every possible error.

The second approach waits for the system to crash during testing, identifies the cause, and adds exception code to handle the crash. Again, the exception code usually reports an error and recovers from the situation. There are also two potential problems with this approach; crashes may not occur until field trials, and it can be very difficult to identify the cause of a crash from the crash dump and any trace logs.

Hence, the first approach is recommended but there is sometimes a fine line between what is believed to be an error and what is actually an error.

Finally, the fault analysis clearly shows that the strategy of subjecting the developed product to 100% testing and not doing any code reviews did not work because seven coding errors were found during field trial that would have been found (in the author's opinion) at a code review. Finding coding errors at the later stages could delay the product as they have to be fixed, retested, and retrialled.

5. CONCLUSIONS

On reflection, it was unwise to adopt the high-risk strategy of 100% testing and not do code reviews. The fault analysis showed too many coding errors found during the latter stages of test and trial, which should have been found earlier.

Therefore, code reviews and testing should both be carried out. They both should be used for their strengths. Code reviews can check for typical coding errors and understand what the code is trying to do. Whereas, testing can check that a product meets its requirements and does what it should do.

The metrics can be used to identify the risks and take appropriate action. For example, any complex code or code that has not been tested, should be reviewed. Likewise, any code that has not been reviewed should be 100% tested.

To be effective, code reviews should involve diligent software engineers with detailed knowledge of the product, the requirements, and available libraries, and general background knowledge of software, and the general subject area.

The testing clearly showed how extremely difficult it was to obtain 100% test coverage across the whole PmAmoProc application. Some modules were 100% tested but the average test coverage for lines and branches were 79.4% and 75.4, respectively. The discrepancy was mainly due to exception handling code. The additional utility to mask the execution code and report a higher coverage may satisfy 100% testing contracts but it does not solve the underlying problem shown by the fault analysis that even exception code has coding errors.

There was some concern that different metrics for the same module showed different % coverage (e.g. 100% line coverage but only 57% branch coverage, 64% branch coverage but only 30% line coverage). This shows the importance of understanding the metrics and how they are generated rather than taking them at face value. It also shows that more than one coverage metric should be used to demonstrate 100% test coverage.

There was a slight incompatibility problem between the HiPath 4000 Manager development environment and the McCabe INTEGRATED QUALITY™ Test tool environment. However, once these were overcome, the tool provided very good support for instrumenting the C++ code, exporting the instrumented code, importing the resultant output from running the instrumented code, and generating the reports.

Finally, product quality can be improved by using the metrics to focus resources on those areas that need reviewing. The project team must decide which metrics are appropriate for their project and what level they should be limited to.

AUTHOR

John Simner is a senior software engineer in Siemens' Design Services at Nottingham, U.K.. He graduated from the University of Birmingham in 1978 with a BSc with Honours Class I in Electronic and Electrical Engineering. He has worked in the Telecommunication Industry for over 25 years, working on real-time embedded and application software in C, C++, and Java. Currently, he is part of a team enhancing a web-based administration and service (A&S) product developed by Siemens Information and Communication Networks. He was part of the first cohort on a MSc course set up between NTU and Roger Andrews Siemens' Head of Engineering. This paper is taken from his MSc. project which developed an application that remotely collected Cordless Telecommunication statistics from HiPath 4000 telecommunication equipment.

ACKNOWLEDGMENTS

I would like to thank Siemens ICN and especially my immediate management; Mr. Roger Andrews, Mr. Paul Erckens and Mr. Graham Underwood for their support. Finally, I owe a lot of gratitude and thanks to my wife Mrs. Janet Simner and children David and Andrew for their love, support, and encouragement whilst doing this paper.

REFERENCES

The following were used as reference material for this paper:

McCABE, 2001. **Using McCabe Test, Version 7.1, Manual.**
Columbia, MD: McCabe Associates.

LAITENBERGER, O. & DEBUAD, J-M, 1998. **An Encompassing Life-Cycle Centric Survey of Software Inspection (ISERN-98-32)**. Fraunhofer Institute for Expermiental Software Engineering, Kaiserslautern, Germany & Lucent Technologies, Naperville, IL

RUSHBY, J., 1991. **Measures and Techniques for Software Quality Assurance**. SRI International, Menlo Park, CA.

WATSON, A.H., 1996. **FastCGIStructured Testing:Analysis and Extensions**. A Dissertation presented to the Faculty of Princeton University in Candidacy for the Degree of Doctor of Philosophy.

WATSON, A.H. & McCABE, T.J., 1996. **Structured Testing:A Testing Methodology Using the Cyclomatic Complexity Metric**. National Institute of Standards and Technology, Gaithersburg, MD. NIST Special Publication 500-235.

APPENDIX A – McCabe Dynamic Analysis – Class Coverage Metrics

This appendix contains the class results of the McCabe Dynamic Analysis of the PmAmoProc application using McCabe INTEGRATED QUALITY™ toolset Version 7.1.

Table 4 to Table 9 show the class coverage metrics for each class in the PmAmoProc application for the following test cases (T1 to T13):

1. With Unity A&S Trace Tool tracing enabled and PmAmoProc tracing disabled, update the list of cordless equipment on the HiPath 4000 switch using the HiPath 4000 Manager Client GUI.

2. With Unity A&S Trace Tool tracing, cordless and feature usage collection all enabled, and PmAmoProc tracing disabled, collect the cordless and feature usage statistics from the HiPath 4000 switch.

3. With Unity A&S Trace Tool tracing enabled and PmAmoProc tracing disabled, update the list of cordless equipment on the HiPath 4000 switch using the now option on the local command line interface.

4. With PmAmoProc tracing disabled and Unity A&S Trace Tool tracing both enabled and disabled, update the list of cordless equipment on the HiPath 4000 switch using the midnight option on the local command line interface.

5. With Unity A&S Trace Tool tracing, PmAmoProc tracing, cordless and feature usage collection all disabled, collect the cordless and feature usage statistics from the HiPath 4000 switch.

6. With Unity A&S Trace Tool tracing and PmAmoProc tracing both disabled, update the list of cordless equipment on the HiPath 4000 switch using the HiPath 4000 Manager Client GUI.

7. With Unity A&S Trace Tool tracing and PmAmoProc tracing both disabled, and cordless and feature usage collection both enabled, collect the cordless and feature usage statistics from the HiPath 4000 switch.

8. With PmAmoProc tracing enabled, update the list of cordless equipment on the HiPath 4000 switch using the HiPath 4000 Manager Client GUI and collect the cordless and feature usage statistics from the HiPath 4000 switch.

9. Using the local command line interface:
    - Invoke PmAmoProc with more than two arguments,
    - Invoke PmAmoProc with an invalid argument,
    - Invoke PmAmoProc with no argument,
    - Invoke PmAmoProc after removing the unlock file,
    - Invoke PmAmoProc after creating installation control file,
    - Invoke PmAmoProc after creating startup control file,

- Invoke PmAmoProc then remove the lock file.

10. With cordless and feature usage collection both enabled, create pm_temp_base_stat.txt and ziel_intern files, and collect the cordless and feature usage statistics from the HiPath 4000 switch.

11. With cordless and feature usage collection both enabled, delete all previous statistic files and station number files the list of cordless equipment on the HiPath 4000 switch using the HiPath 4000 Manager Client GUI, and collect the cordless and feature usage statistics from the HiPath 4000 switch.

12. With cordless and feature usage collection both enabled, delete the zausl control file (PmAmoProc.zausl), and collect the cordless and feature usage statistics from the HiPath 4000 switch.

13. With cordless and feature usage collection both enabled, create the collection stop files in the file transfer directory, and collect the cordless and feature usage statistics from the HiPath 4000 switch. After a few minutes, delete the collection stop files.

The Unity A&S Server Trace Tool is a proprietary trace tool used in the HiPath 4000 Manager. PmAmoProc tracing is a local tracing utility within the PmAmoProc application. The cordless collection and feature usage collection are controlled by local control files.

The tables include the percentage coverage for three metrics; Cyclomatic Complexity, Module Design Complexity, and branches.

A module's Cyclomatic Complexity (v(G)) is the minimum number of linearly independent paths through the module (see McCabe 2001, p51-53).

A module's Design Complexity (iv(G)) is the number of paths with calls to other modules (see McCabe 2001, p55-57).

Branch coverage shows how many of the exits from branches were executed.

Table 4 - Class Coverage Metrics for BaseStationStat

% Coverage	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
v(G)	0.0	36.7	36.7	36.7	36.7	36.7	59.0	62.8	62.8	62.8	62.8	62.8	62.8
iv(G)	0.0	38.0	38.0	38.0	38.0	38.0	62.8	66.7	66.7	66.7	66.7	66.7	66.7
Branches	0.0	56.2	56.2	56.2	56.2	56.2	70.4	73.0	73.0	73.0	73.0	73.0	73.0

Table 5 - Class Coverage Metrics for BaseStationStatConfig

% Coverage	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
v(G)	0.0	44.3	44.3	44.3	44.3	44.3	60.7	60.7	60.7	60.7	60.7	60.7	60.7
iv(G)	0.0	46.7	46.7	46.7	46.7	46.7	63.9	63.9	63.9	63.9	63.9	63.9	63.9
Branches	0.0	64.7	64.7	64.7	64.7	64.7	74.9	74.9	74.9	74.9	74.9	74.9	74.9

Table 6 - Class Coverage Metrics for CardStatSLC16

% Coverage	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
v(G)	0.0	34.2	34.2	34.2	34.2	34.2	55.8	55.8	55.8	55.8	55.8	55.8	55.8
iv(G)	0.0	36.5	36.5	36.5	36.5	36.5	61.4	61.4	61.4	61.4	61.4	61.4	61.4
Branches	0.0	57.2	57.2	57.2	57.2	57.2	70.7	70.7	70.7	70.7	70.7	70.7	70.7

Table 7 - Class Coverage Metrics for FeatUsageStat

% Coverage	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
v(G)	0.0	44.1	44.1	44.1	58.0	58.0	71.7	71.7	71.7	71.7	71.7	71.7	80.1
iv(G)	0.0	45.8	45.8	45.8	61.3	61.3	76.2	76.2	76.2	76.2	76.2	76.2	85.1
Branches	0.0	69.6	69.6	69.6	78.6	78.6	87.1	87.1	87.1	87.1	87.1	87.1	94.9

Table 8 - Class Coverage Metrics for PmAmoIf

% Coverage	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
v(G)	11.3	14.6	14.6	24.6	26.5	26.5	27.8	27.8	27.8	27.8	27.8	28.7	28.7
iv(G)	12.4	15.6	15.6	26.2	28.2	28.2	29.5	29.5	29.5	29.5	29.5	30.4	30.4
Branches	35.6	52.4	52.4	61.6	63.8	63.8	65.4	65.4	65.4	65.4	65.4	66.9	66.9

Table 9 - Class Coverage Metrics for PmAmoProc

% Coverage	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13
v(G)	12.6	19.5	19.5	28.8	33.9	33.9	40.8	40.8	40.8	40.8	43.4	44.4	45.2
iv(G)	12.6	19.5	19.5	28.9	33.9	33.9	40.8	40.8	40.8	40.8	43.4	44.4	45.2
Branches	26.9	55.3	55.3	60.9	67.2	67.2	68.1	68.1	68.1	68.1	70.0	71.0	72.2