An Industrial Case Study of Structural Testing Applied to Safety-critical Embedded Software

Jing Guan
Information and Software Engineering
George Mason University
Fairfax, VA 22030, USA
jguan@gmu.edu

Jeff Offutt
Information and Software Engineering
George Mason University
Fairfax, VA 22030, USA
offutt@ise.gmu.edu

Paul Ammann
Information and Software Engineering
George Mason University
Fairfax, VA 22030, USA
pammann@gmu.edu

ABSTRACT

Effective testing of safety-critical real-time embedded software is difficult and expensive. Many companies are hesitant about the cost of formalized criteria-based testing and are not convinced of the benefits. This paper presents the results of an industrial case study that compared the normal testing at a company (manual functional testing) with testing based on the logic-based criterion of correlated active clause coverage (CACC). The evaluation was performed during the testing of embedded, real-time control software that has been deployed in a safety-critical application in the transportation industry[1]. We found in our study that the test cases generated to satisfy the CACC criterion detected major safety-critical faults that were not detected by functional testing. We also found that the cost required for CACC testing was not necessarily higher than the cost of functional testing. There were also several faults that were found by the functional tests that were **not** found by CACC tests.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; J.7 [**Software Engineering**]: Industrial control

General Terms

Measurement, Experimentation, Verification

Keywords

Software testing, embedded software, industrial case study

[1]To protect its confidentiality, the company is not allowing publication of any details that could identify the product or company. For the same reason, the first author's professional affiliation has been omitted.

ISESE'06, September 21–22, 2006, Rio de Janeiro, Brazil.

1. INTRODUCTION

Embedded software is part of a larger hardware device or other system. Although there are many types of embedded software systems, much of it is used to control hardware devices. *Control embedded software* is often real-time, safety-critical, and deployed with little or no direct interface to users. It is quite common for embedded software to be built by companies whose primary focus is **not** software, and reasonably common for embedded software to be built by hardware engineers (most often electrical or mechanical). This is sometimes required because embedded software usually requires extensive knowledge of the hardware devices they are being embedded into.

Embedded software is sometimes tested in inefficient and very expensive ways. Although use of simulation is common, it is by no means universal. Our experience includes a professional career as an electrical engineer and software engineer for companies that develop hardware devices with embedded software (first author), significant consulting experience with embedded software (second author), and extensive experience educating professional software developers and testers in all segments of the industry (second and third authors). In our experience, software testing of embedded software is often done by hand, on the actual hardware, and typically without the tester having guidance, knowledge or access to software tool support. At the same time, the typical safety requirements means that testing must be very effective, so engineers wind up devoting huge amounts of time to the testing. These observations are supported by a recent survey of Swedish software companies by Grindal et al. [7].

There are many reasons why testing is not more sophisticated. One possible reason is that program managers are unconvinced that sophisticated test criteria can help, and suspicious that they will be too expensive or difficult to apply to "real" software outside of university labs. The goal of this project was to ask whether a particular test criterion can be **useful** and **cost-effective** in practice. This is a practical industrial case study, not a rigorously controlled experiment.

This paper presents a case study of testing real-time control software that is embedded into a safety-critical application in the transportation industry[1]. Two sets of tests were used. The manual tests that the engineers at the company created ("manual functional testing - MFT") were compared with tests generated from the code to satisfy a high-end

logic-based criterion (CACC [1], similar to the masking form of the well known MCDC [2]). This study compared CACC testing with MFT and found that the CACC test cases detected important faults that were not detected by manual functional testing, and that would be very difficult to detect by any other testing technique. On the other hand, several faults were found by MFT that were not found by CACC.

Moreover, the cost of applying CACC in this study was actually less than the cost of the manual functional tests (omitting training of the tester). The cost result is less convincing because of the very different way in which the logic testing criterion was applied. More automation was employed and simulation was used to reduce human effort.

Section 2 of this paper describes logic-based testing in general and describes CACC in particular. Section 3 introduces the industrial application we tested and describes the tests. Section 4 presents the results from our case study and Section 5 summarizes our conclusions and suggests future work.

2. STRUCTURAL LOGIC-BASED TESTING

Logical expressions are common to almost every type of software artifact, including program source code, finite state machines, and formal specifications. Because they are so common, easy to formalize, and easy to process automatically, several test criteria have been defined that are based on logical expressions. When logic-based testing is based on program source code, it is called *structural logic-based testing*. While logic coverage criteria have been known for a long time, their use has been steadily growing in recent years. One major cause for their use in practice has been because the US Federal Aviation Administration (FAA) requires that one of the logic coverage criteria, Modified Condition Decision Coverage (MCDC) [3], be used for safety critical parts of the avionics software in commercial aircraft [5, 9]. A number of other criteria exist, some of which are equivalent to MCDC and others of which are very similar. In fact, a careful study of the published literature on the subject reveals that the same ideas have been repeated several times in different contexts [2, 3, 4, 6, 8, 10, 11].

2.1 Correlated Active Clause Coverage

One such logic-based test criterion, introduced by Ammann and Offutt [1], is called Correlated Active Clause Coverage (CACC). Its goal is to test individual clauses within logical expressions and it yields tests that are identical to tests developed to satisfy the masking form of MCDC [2].

CACC formalizes logical expressions in a common mathematical way. A *predicate* is an expression that evaluates to a boolean value. A simple example is: $((a > b) \vee C) \wedge p(x)$. Predicates may contain boolean variables, non-boolean variables that are compared with relational operators, and calls to functions that return a boolean value, all three of which may be joined with one or more of the logical operators negation ($\neg$), and ($\wedge$), or ($\vee$), implication ($\rightarrow$), exclusive or ($\oplus$), and equivalence ($\leftrightarrow$). A predicate that does not contain any of the logical operators is called a *clause*. The predicate above has three clauses, $(a > b)$, C, and $p(x)$.

CACC is articulated in terms of test requirements. *Test requirements (TR)* are specific elements of software artifacts that must be satisfied or covered. Test requirements can be described in terms of a variety of software artifacts, including the source code, design components, specification modeling elements, or even descriptions of the input space. *Test specifications* are specific descriptions of test cases, often associated with or derived from test requirements.

These two terms allow a straightforward definition for a test coverage criterion. A *test criterion* is a rule or collection of rules that impose test requirements on a set of test cases. That is, the criterion describes the test requirements in a complete and unambiguous manner.

The intent of CACC is to force the tester to evaluate individual clauses as well as entire predicates. To test individual clauses in such a way that they affect the predicate, it is necessary to give specific values to the other clauses in the predicate. Thus, we want the value of the predicate to directly depend on the clause being tested. For convenience of expression, the "clause that we want to test" is called the *major clause*.

Definition 1 Determination: *Given a clause c_i in predicate p, called the major clause, c_i* determines *p if the remaining minor clauses $c_j \in p$, $j \neq i$, have values such that changing the truth value of c_i changes the truth value of p.*

This definition explicitly does **not** require that the major clause and the predicate have the same value ($c_i = p$). This issue has been left ambiguous by previous definitions, and similar criteria (including MCDC) have sometimes been taught as requiring that the predicate and the major clause must have the same value. This interpretation is not practical. When the negation operator is used, for example, if the predicate is $p = \neg a$, it becomes impossible for the major clause and the predicate to have the same value. Algorithms for finding truth assignments to non-major (called minor) clauses to ensure that the major clause determines the value of the predicate have been published [1].

CACC uses determination by requiring that major clauses determine the value of the predicate. In the following definition, P is a set of predicates that are derived from some program artifact such as the program source, an FSM, or a formal specification. C_p is the set of clauses in P.

Definition 2 Correlated Active Clause Coverage (CACC): *For each $p \in P$ and each major clause $c_i \in C_p$, choose minor clauses c_j, $j \neq i$ so that c_i determines p. TR has two requirements for each c_i: c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j must cause p to be true for one value of the major clause c_i and false for the other, that is, it is required that $p(c_i = true) \neq p(c_i = false)$.*

Consider the example $p = a \wedge (b \vee c)$. For clause a to determine the value of p, the expression $b \vee c$ must be true, which can be done in one of three ways: b true and c false, b false and c true, and both b and c true. So, it would be possible to satisfy Correlated Active Clause Coverage with respect to clause a with the two test requirements: $\{(a = true,\ b = true,\ c = false),\ (a = false,\ b = false,\ c = true)\}$. There are other possible sets of test requirements with respect to a, as enumerated in the following partial truth table. There are nine possible truth assignments that will satisfy CACC for a, by choosing one test requirement from rows 1, 2 and 3, and another from rows 5, 6 and 7.

	a	b	c	$a \wedge (b \vee c)$
1	T	T	T	T
2	T	T	F	T
3	T	F	T	T
5	F	T	T	F
6	F	T	F	F
7	F	F	T	F

Although no commercial tool is available, most of the steps for applying CACC can be automated. Predicates can be extracted automatically, truth assignments could be found for minor clauses for determination, and truth assignments could be found for the major clauses to satisfy CACC. The step of finding values for inputs to result in the necessary truth assignments for clauses is more difficult to automate. The case study reported in this paper applied CACC to predicates derived from the program source.

3. THE TARGET SYSTEM AND TESTS

The control system used in this case study is a very complicated collection of interacting state machines and algorithms that was completed and deployed in 2003. It interacts with a number of hardware devices, including a battery, generator, several sensors and a wireless communicator. The software uses several kinds of interrupts to perform real-time control.

The control system has five different modes and manages different actions in each mode. For example, after the control system is powered on, the system enters Mode $M1$. Certain conditions cause the system to switch from one mode to another.

The software was written in C by the first author as part of her work assignment as a full-time employee of the company, and consists of 12 files and 90 functions ($>$ 3000 LOC). The software contains 70 predicates. 50 predicates had one clause, 17 predicates had two, and 3 predicates had three clauses. The software used was a complete, pre-production version.

Generally speaking, functional testing derives test cases from the software requirements specification. The manual functional tests were created by a test engineer at the company. They were created by hand, following a design validation test plan. In our study, the company testers applied functional testing in their usual way, running tests directly on the target hardware. That is, they were run without using simulation. This paper refers to this process as "manual functional testing," or "MFT."

CACC was applied to the predicates that appeared in the program source. No tool was available to support the construction of CACC tests, so they were created by hand analysis of the software. Arbitrary values were also selected by hand to satisfy the required truth assignments. These predicates resulted in 134 CACC tests. Following is an example of how test cases were designed to satisfy the CACC criterion. One predicate has three clauses (the variable names have been change because of company confidentiality requirements):

```
((A < 144) && (B < 270) && (C < 880))
```

Table 1 lists the test requirements needed to satisfy CACC coverage and the resulting truth value of the predicate. Rows 1 and 2 contain test requirements for clause ($A < 144$), rows 1 and 3 contain test requirements for clause ($B < 270$), and rows 1 and 4 contain test requirements for clause ($C < 880$),

	A $<$ 144	B $<$ 270	C $<$ 880	Pred
1	T	T	T	T
2	F	T	T	F
3	T	F	T	F
4	T	T	F	F

Table 1: Example of test requirements to satisfy CACC.

1	A = 140, B = 250, C = 800
2	A = 140, B = 250, C = 900
3	A = 140, B = 300, C = 800
4	A = 145, B = 250, C = 800

Table 2: Test values to satisfy CACC.

Table 2 lists the test case values that were selected to satisfy the test requirements from Table 1.

One common problem with testing embedded control systems is that it is very difficult to **control** all the parameters that affect the system. For example, the software reads twenty analog inputs from an eight channel Analog-To-Digital converter, which means several analog inputs may share one common channel. An analog MUX chip is used to control the analog-to-digital converter to switch among different analog inputs in one single channel. The switching time is usually less than 1 milli-second, which makes it impossible for the tester to identify or control the specific analog input to the microprocessor.

There is also a limitation on one variable (here called P). Because of hardware limitations, it was only possible to set P to be up to 85 during testing, but P could get as high as 120 in field usage.

These problems were solved during CACC testing by simulation. All the analog inputs from hardware can be replaced with simulation software statements that explicitly set the needed values. For example, the software interfaces with the hardware to read a value for P with the following statement:

```
P = ((analogP * x/1024 - 0.25) * 125/2) * y + z;
```

This was simulated during testing by modifying the program:

```
P = 120;
```

A number of timer counter values were used as test values in CACC testing, but timers are set by the real-time timer interrupt routines during execution. To monitor the timer test values as well as make the system run in real time, we output the timer counter values to a computer screen through a serial port, then supplied the next inputs at the appropriate time.

The CACC tests were run on the target hardware in a simulation environment, which provided the software with inputs and collected results, including output messages. Each test case was run via a test script that set the initial conditions of the system, called the simulation program, launched the software, and displayed the results. Additional routines were written to allow the tester to start the software in a particular mode, choose the next test case, and run in "normal mode."

Fault	MFT	CACC
F1	X	X
F2	X	X
F3	X	X
F4	X	
F5	X	
F6	X	
F7	X	
F8	X	
F9		X
F10		X
F11		X
F12		X
F13		X
F14		X
F15		X
F16		X
F17		X
F18		X
F19		X

Table 3: Faults found by MFT and CACC.

4. CASE STUDY AND RESULTS

This section describes the MFT and CACC testing results, and discusses the implications of the results in regards to the relationship between CACC coverage and software safety. Data from the difficulty of satisfying the CACC criterion is also provided. Table 3 gives an abstract summary of the 19 faults found by the two sets of tests.

4.1 Manual Functional Testing

A test engineer performed the manual functional testing directly on the target hardware. Simulation was not used and the results were checked by hand. Eight faults were found. All the failures were detected by visual inspections of the behavior of the hardware. Faults $F1$, $F2$ and $F3$ were found by both types of tests. Faults $F4$ and $F5$ were related to user interface issues that were not addressed by the CACC tests. Faults $F6$, $F7$ and $F8$ were found by running the hardware for a long period of time. Structural tests based on source are seldom effective at finding these types of faults. They are usually addressed only by system level tests based on domain knowledge of the application.

4.2 CACC Testing

The CACC tests found three of the same faults that MFT found ($F1$, $F2$, and $F3$), plus eleven additional faults. Faults $F9$ and $F10$ were logic mistakes in multi-clause predicates and seem very unlikely to have been found by any criterion less stringent than CACC (or MCDC). Faults $F11$ and $F12$ were in the software design, $F13$ could not be found without simulation, and $F14$ through $F19$ required analysis of outputs that were difficult or impossible to observe without using simulation. At least one of the faults related to multi-clause predicates, $F9$, was safety related and could easily have caused a catastrophic failure during operation, possibly involving loss of life.

Two faults were a result of late changes in the specifications that were **not** implemented in all appropriate locations in the software. Fault $F9$ is outlined here–in abstract form because of company confidentiality requirements.

In normal operation, the unit goes to different modes based on two inputs–here called $I1$ and $I2$. A specific value for variable X puts the unit into mode $M1$, while a value of another variable Y puts the unit into mode $M2$. The following decision encodes this operation:

Decision Block 1:

```
if (X >= 50)
   modeM1();
else if (Y > 18)
   modeM2();
else if (Y <= 18)
   modeM3();
```

In another location in the program, the unit wakes up from sleep mode to normal operation at the same X decision point (50):

Decision Block 2:

```
if ((X < 50) && (Y <= 18))
   modeM3();
```

Once the system goes to normal operation, it makes decisions based on Decision Block 1.

After the initial implementation, the design was changed to use 45 as the decision point instead of 50, resulting in the following change within Decision Block 1:

```
if (X >= 45)
   modeM1();
```

The software fault was that Decision Block 2 was **not** updated with the change to 45. In the manual functional tests, when 45 was applied to the sleeping unit, Y was also greater than 18, so the unit was correctly woken up. But it was woken because of the value of Y, that is, the clause involving X was not well validated. Then when Decision Block 1 was reached, 45 put the unit into mode $M1$, so the tester erroneously concluded that the predicates were correct. This fault is very unlikely to be caught unless a test criterion that explicitly forces multiple clauses to be tested independently is used.

Two other faults, $F11$ and $F12$, were a direct result of mistakes in the design specification. For example, in operation mode, a digital-to-analog converter is loaded with the value P to adjust the output voltage for battery charging. The software changes the output voltage by increasing a variable P based on the predicate "((Q >= 14.7) && (P < 880))." The specification said to initialize P to 900, which in turn sets the value of Q to 10.5, and then the value for Q should be increased slowly to 14.7 by decreasing P. When Q gets to 14.7, P should be much less than 880, so the condition can be satisfied and the voltage is adjusted. However, when another variable R < 10.5, Q is immediately set to 14.7. If this happens at the beginning of the process, right after P is initialized to 900, the clause ($P < 880$) will not be satisfied, and then the value for Q will not be decreased. This fault could cause the unit to keep on increasing above the required limit, resulting in high voltage that would damage the battery. During MFT, a fully charged battery was used to supply voltage, so the fault was not detected. The CACC tests, on the other hand, required individual clauses to be evaluated, forcing the fault to be found.

An interesting result is that a number of faults were found that could not have been observed without using simulation.

As an example, the software constantly samples a sensor. The variable that stores this number is declared as an unsigned char. By using simulation in CACC testing, it was found that the value could be over 255, which could cause it to "roll over" to 0, in which case the software would make a wrong decision based on this incorrect input. Manual functional testing never created this condition, so the fault was not detected. In fact, this fault could only be created on the actual hardware during a catastrophic crash, which could not be caused during the company's manual functional testing.

4.3 CACC Coverage and Software Safety

Based on the number of faults detected by CACC testing, including at least one that was safety critical, we conclude that CACC testing improved the safety of the software. At the same time, CACC coverage required a lot of test cases – more than the company's MFT did.

Some features of the software were not included in the software specification and therefore manual functional tests were not generated. CACC testing was based on the implementation, and therefore can help verify the completeness of the specification.

Timers are crucial factors in safety-critical real-time embedded software. Incorrect timer events can cause the system to behave inconsistently. Some timing errors were hard to detect by manual functional testing without simulation. CACC testing has the ability to track the value of every timer used in the software.

The software that was tested has a complex logic mechanism requiring precise, full understanding and customized testing. Testing complicated logic can be more effective when tests are based on the implementation, because the implementation often includes details that were omitted from the specification.

4.4 Difficulty of Achieving CACC

The CACC tests were generated by the first author, who carefully kept track of the time spent designing and generating the tests. Note that no tools were used to help determine coverage and this was the experimenter's first experience in applying CACC testing outside of the classroom. That is, this time can be considered an "upper bound" on the time needed in practice. The experimenter spent three hours extracting the logic predicate, six hours designing the CACC test cases, ten hours generating test scripts, seven hours running tests and six hours analyzing test results. This was a total of thirty-two hours used to achieve CACC coverage. Because all steps were done by hand, this can be considered an upper limit of the time. Tool support and experience could cut the amount of time used.

Another noteworthy part of the data was in the number of predicates, which is directly related to the cost of applying CACC. The software had a total of 70 predicates, each of which had to be analyzed. It should be clear that predicates with more clauses are harder to construct tests for than predicates with few clauses. In fact, satisfying CACC on predicates with only one clause is equivalent to satisfying branch coverage. Of those 70 predicates, 50 (71.4%) had only one clause. Another 17 (24.2%) had only two clauses, and only 3 predicates had three clauses (4.3%). These numbers certainly make the cost of satisfying CACC less than what a test manager might fear. On the other hand, simpler techniques such as branch testing and multiple condition testing (testing all combinations of truth values for predicates) may well be just as effective on predicates with fewer clauses. We have observed similar percentages in larger counts of open source software, leading us to believe that it is possible that multi-clause predicates are relatively few. This assumption is implicitly made by the commercial tool Agitator, which automatically generates tests to satisfy full multiple condition testing.

Manual functional testing was carried out by an experienced tester at the company, who took two days to validate the test plan, and four days to run all the tests, for a total of 48 hours. This time was increased by stress testing. For example, one test was to check if the hardware goes into sleep mode after running in one of the operation modes for fifteen hours. In MFT, the tester had to spend fifteen hours to observe if the unit behaved correctly. In CACC testing, a test case with a similar goal was run by initializing this operation timer to be 14 hours 55 minutes. The test then finished in 5 minutes.

Despite the fact that project managers will often say that "criteria-based testing takes too much time," this study found that CACC testing used significantly **less** time than MFT.

4.5 Discussion of Validity

It is naturally harder to control for experimental validity in an industrial case study. This research was an in-depth study, but only used one software application from one company. Therefore it is at best risky to draw general conclusions (external validity). However, our intent was to modestly determine whether CACC can be useful and cost-effective in an industrial setting, which does not require external validity. We have no data about whether this testing method will work equally well on different or larger software applications, but can think of no reason why it would not.

The study unavoidably had two independent variables, the method used to **generate** tests (MFT and CACC) and the method used to **execute** tests (on the hardware and with simulation). This is common with industrial case studies. The experiment cannot be completely designed *a priori*; researchers must accept the current situation. The effects were separated as much as possible in our analysis. Of the 11 faults that CACC found and MFT did not, five could have been found with or without simulation. So even if MFT also used simulation, they would probably not have been found. It is possible that the MFT tests would have found some of the other six faults, but we cannot be sure. Even so, this does not affect the general conclusions.

We addressed internal validity and reduced bias in several ways. The same software was tested by both sets of tests and different people created and ran both tests. The MFT tester had more experience than the CACC tester, which introduced a slight bias **against** CACC, thereby strengthening the internal validity of the results.

5. CONCLUSIONS

This paper presents results from an industrial case study of logic-based testing applied to safety-critical embedded software, and compared the logic-based tests with tests created by testers at the company using manual functional testing. The intent was to determine whether a high-end logic-based test criterion could be successful in a practical in-

dustrial setting. The specific logic-based test criterion used was correlated active clause coverage (CACC). Tests to satisfy the CACC criterion found several significant faults that were **not** found by MFT. Our analysis indicates that some of these faults were unlikely to have been detected with weaker test criteria. Thus, we assert that this study is strong evidence that high-end logic-based test criteria can successfully increase the reliability of industry software.

Automation and simulation was also found to be necessary to find faults. It should also be noted that CACC by itself is clearly not enough for industrial use; as pointed out in Section 4.1, five faults were found by MFT that were not found by the CACC tests. Finally, because of the close connection between the hardware and the software, testing real-time embedded software requires knowledge of both the software and hardware.

Structural logic-based testing criteria are not often used in practice; this case study gives evidence that they can be useful and cost effective for safety-critical embedded software. We hope that the positive results on this real-world example will help convince software developers that structural, criteria-based testing are economically beneficial.

These results are based on one application, and in the future it would certainly be helpful to repeat similar studies on other software applications in other environments. Ideally, a rigorous, carefully designed and controlled experiment on multiple industrial software applications would be carried out. However this kind of study is very difficult, extremely expensive, and vanishingly rare. Data for practical applications of theoretical engineering ideas usually must be accumulated over a period of time by multiple researchers in multiple studies. Only then can software development managers be confident that the techniques in question will be worth the investment.

6. ACKNOWLEDGMENTS

We would like to anonymously thank the company for giving us access to their software and results. A brief extract of this work appeared as a fast abstract at ISSRE 2003.

7. REFERENCES

[1] P. Ammann, J. Offutt, and H. Huang. Coverage criteria for logical expressions. In *Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 99–107, Denver, CO, November 2003. IEEE Computer Society Press.

[2] J. Chilenski and L. A. Richey. Definition for a masking form of modified condition decision coverage (MCDC). Technical report, Boeing, Seattle, WA, 1997. http://www.boeing.com/nosearch/mcdc/.

[3] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, September 1994.

[4] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[5] A. Dupuy and N. Leveson. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In *Proceedings of the Digital Aviations Systems Conference (DASC)*, October 2000.

[6] K. Foster. Error sensitive test case analysis. *IEEE Transactions on Software Engineering*, 6(3):258–264, May 1980.

[7] M. Grindal, J. Offutt, and J. Mellin. On the testing maturity of software producing organizations. In *Testing: Academia & Industry Conference - Practice And Research Techniques (TAIC / PART 2006)*, Windsor, UK, August 2006. IEEE Computer Society Press.

[8] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Software Testing, Verification, and Reliability*, 13(1):25–53, March 2003.

[9] RTCA-DO-178B. Software considerations in airborne systems and equipment certification, December 1992.

[10] S. A. Vilkomir and J. P. Bowen. Reinforced condition/decision coverage (RC/DC): A new criterion for software testing. In *Proceedings of ZB2002: 2nd International Conference of Z and B Users*, pages 295–313, Grenoble, France, January 2002. Springer-Verlag, LNCS 2272.

[11] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *IEEE Transactions on Software Engineering*, 20(5):353–363, May 1994.