An Empirical Evaluation of the MC/DC Coverage Criterion on the HETE-2 Satellite Software*

Arnaud Dupuy, Alcatel, France
Nancy Leveson, MIT, USA

Abstract

In order to be certified by the FAA, airborne software must comply with the DO-178B standard. For the unit testing of safety-critical software, this standard requires the testing process to meet a source code coverage criterion called Modified Condition/Decision Coverage. This part of the standard is controversial in the aviation community, partially because of perceived high cost and low effectiveness. Arguments have been made that the criterion is unrelated to the safety of the software and does not find errors that are not detected by functional testing. In this paper, we present the results of an empirical study that compared functional testing and functional testing augmented with test cases to satisfy MC/DC coverage. The evaluation was performed during the testing of the attitude control software for the HETE-2 (High Energy Transient Explorer) scientific satellite (since that time, the software has been modified). We found in our study that the test cases generated to satisfy the MC/DC coverage requirement detected important errors not detectable by functional testing. We also found that although MC/DC coverage testing took a considerable amount of resources (about 40% of the total testing time), it was not significantly more difficult than satisfying condition/decision coverage and it found errors that could not have been found with that lower level of structural coverage.

1 Introduction

To be certified by the FAA, aviation software must satisfy a standard labelled DO-178B [4]. Software development processes are specified in this standard for software of varying levels of criticality. With respect to testing, the most critical (Level A) software, which is defined as that which could prevent continued safe flight and landing of the aircraft, must satisfy a level of coverage called Modified Condition/Decision Coverage (MC/DC).

The requirement for MC/DC coverage has been criticized by some members of the aviation industry as being very expensive but not very effective in finding errors, particularly safety-critical errors. None of these complaints, however, are backed up with data as companies are, with good reason, unwilling to publish details of their testing process and results.

To shed some light on the issue, we performed an empirical evaluation of the criterion on the attitude control software of the HETE-2 (High Energy Transient Explorer) scientific satellite being built by the MIT Center for Space Research for NASA [2]. Our study compared functional testing and functional testing augmented with test cases to satisfy MC/DC coverage. Although one data point is inadequate to come to definitive conclusions, it is better than the current arguments based on no or little publicly available data. Additional studies should be done to verify our results. In addition, our use of real aerospace software allows conclusions related to the unique features often found in such software and applications.

In the next two sections, we provide a brief description of MC/DC and the software that was tested. Then we describe the design of the study and present an analysis of the results.

2 Structural Testing using Modified Condition/Decision Coverage

Software module testing is used to verify both that the software does what it is supposed to do and that the soft-

*This paper was presented at DASC (Digital Aviation Systems Conference) in Phildelphia, Oct. 2000 and is included in the proceedings.

ware does not do what it is not supposed to do [8]. To meet this goal, there exist two testing strategies. *Blackbox* testing ignores the structure of the source code and derives test cases only from the specification in order to detect anomalous software behavior. *Whitebox* or structural testing, on the other hand, takes advantage of knowledge of the structure of the source code to design test cases [6].

In the rest of the paper, we use the following definitions:

Condition A condition is a leaf-level Boolean expression (it cannot be broken down into a simpler Boolean expression).

Decision A decision is a Boolean expression that controls the flow of the program, for instance, when it is used in an **if** or **while** statement. Decisions may be composed of a single condition or expressions that combine many conditions.

Structural testing criteria have been defined that describe the level of coverage of the code:

Statement Coverage: Every statement in the program has been executed at least once.

Decision Coverage: Every point of entry and exit in the program has been invoked at least once, and every decision in the program has taken all possible outcomes at least once.

Condition/Decision Coverage: Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, and every decision in the program has taken all possible outcomes at least once.

Modified Condition/Decision Coverage: Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken on all possible outcomes at least once, and each condition has been shown to affect that decision outcome independently. A condition is shown to affect a decision's outcome independently by varying just that decision while holding fixed all other possible conditions.

The condition/decision criterion does not guarantee the coverage of all conditions in the module because in many test cases, some conditions of a decision are masked by the other conditions. Using the modified condition/decision criterion, each condition must be shown to be able to act on the decision outcome by itself, everything else being held fixed. The MC/DC criterion is thus much stronger than the condition/decision coverage criterion, but the number of test cases to achieve the MC/DC criterion still varies linearly with the number of conditions n in the decisions. Therefore, the MC/DC criterion is a good compromise for whitebox testing: (1) it insures a much more complete coverage than decision coverage or even condition/decision coverage, but (2) at the same time it is not terribly costly in terms of number of test cases in comparison with a total coverage criterion [5].

The controversy over the MC/DC coverage requirement revolves around cost and effectiveness issues. It should also be noted that the technique does not relate directly to requirements or safety considerations: although the MC/DC criterion is imposed to ensure that the software is safe, the testing is not related to the system or software safety requirements or constraints.

3 Relevant Aspects of the HETE-2 Attitude Control Software

Our case study used the HETE-2 (High Energy Transient Explorer), a science mini-satellite developed at MIT, and more precisely, its Attitude Control System (ACS) software [2]. A failure of the ACS can cause the loss of the satellite or at least a complete failure of the mission. Although this software is not part of an aeronautics system, its requirements, constraints, and safety issues are similar enough to avionics software for the case study to be relevant to the environment in which DO-178B is usually applied.

A first HETE satellite was launched in November 1996, but it was lost due to the failure of its Pegasus XL launcher. This case study was run during the development of a second satellite, HETE-2, with the same scientific goals. Its mission is the study of the astrophysical events known as gamma ray bursts. Gamma ray bursts are high-energy transients in the gamma range that seem to be isotropically distributed in the sky. They last from a millisecond to a few hundreds of seconds and involve a huge amount of energy. HETE-2 is expected to detect the bursts, locate them, and capture their spectral characteristics.

The spacecraft carries several instruments used for the mission: four gamma ray telescopes, two wide-field X-ray cameras, two soft X-ray cameras, and two optical cameras. The optical cameras provide the Attitude Control System with the drift rates during orbit nights.

The ACS uses two types of sensors to determine the attitude of the spacecraft: sun sensors and magnetometers. The sun sensors allow the ACS to compute the attitude of the spacecraft with respect to the sun while the magnetometers allow the ACS to determine the spin vector of the spacecraft. The spacecraft attitude is modified using three torque coils and one momentum wheel. The communication between the sensors/actuators and the computer is made via a serial bus, called the AUX bus.

The ACS software is written in C and compiled using the GCC GNU compiler under a Sun Solaris environment. The software contains approximately 6000 lines of source code. Ultimately, the software will run on the spacecraft's on-board transputer, and the object code for this processor will be generated by a cross compiler. No simulation facility is supported on the transputer, so the module testing had to be done in the Unix environment and only functional testing and system testing will take place on the transputer itself. Because DO-178B requires module testing to be performed in the target environment, the testing does not conform with the standard on this point. Testing on HETE does not protect against cross-compiler bugs, but in this case, host testing was the only feasible way to conduct the module testing process and should not affect this study of the efficacy of the DO-178B testing procedure.

The ACS controls the deployment sequence from the moment the satellite is released by the rocket (with solar paddles stowed and with a tumbling attitude) until the moment it reaches its final orbit configuration (with paddles deployed and axis stabilized). During the operations phase, the ACS is crucial for the correct operation of the instruments, for power balance, and for thermal balance. The ACS requirements are divided into orbit-day requirements and orbit-night (eclipse time) requirements.

In order to perform efficiently all the different operations for which it is responsible, the ACS is divided into ten different modes: modes 0 to 6 are used during the deployment sequence while modes 7 and 8 are activated alternately during the operations phase. Mode 9 is a backup ground command mode. The progression in the succession of modes (from mode 0 where the spacecraft is tumbling right after it is released by the rocket to modes 7–8 where the payload can operate) corresponds to an improvement in the spacecraft stabilization.

The first four modes use only the magnetic torque coils as actuators. The control laws that are implemented by the ACS software in these modes are quite simple: Their goal is to dampen most of the rotation speed transmitted to the satellite by the spacecraft so that it acquires a rotational stiffness that allows it to stay aligned with the sun.

The next modes bring the momentum wheel into play in order to stabilize the spacecraft finely (with very low drift rates). These algorithms are more complex, in particular a Kalman filter is used. The deployment of the solar paddles also occurs in this part of the deployment sequence, when the satellite is in a steady position, facing the sun.

The progression through the modes need not be linear. Nominally, the ACM goes through the deployment sequence (mode 0 to mode 6) and then toggles between mode 7 (orbit day) and mode 8 (orbit night) during operations. But a set of parameters is constantly monitored, and if one grows past its corresponding threshold, the ACS switches back to the mode that is optimized to fix this parameter (taking into account whether it is orbit day or orbit night). As a result, the mode switching logic contains many variables and paths. In addition, many of the mode switching conditions involve required time delays.

The testing of the ACS software involves checking the following:

- The switching logic is correctly implemented (the switching between the modes occurs when the spacecraft is in the expected configuration).
- The behavior of each mode is correct (each mode performs the task it is designed for and does not corrupt any other parameters).
- The spacecraft meets the ACS requirements while it is on station (after it has gone through the acquisition sequence, the spacecraft maintains a correct attitude so that the other subsystems can perform normally).

4 Design of the Study and The Testing of the Software

For Level A software, DO-178B requires functional testing augmented with the test cases required to guarantee coverage according to the MC/DC criterion. Testing attitude control systems has always been a problem in the space industry as it is virtually impossible to control all the parameters that affect the system, such as the orientation of the sun, the components of the magnetic field, gravity, the small perturbations that affect the spacecraft in orbit, etc. For this reason, the testing of the ACS hardware and the ACS software are decoupled. Each hardware item is verified on its own, and a simulation environment is created to provide the software with the information it expects and to collect the commands it outputs.

For HETE-2, a complete simulation environment was available for testing the ACS. The simulation environment can feed the ACS software with all the environment parameters corresponding to the position of the satellite (sun direction or orbit night, magnetic field, disturbance torque, etc.). It can also simulate the dynamics of the spacecraft: given initial conditions, actuator commands, and environment torques, the state of the satellite (rotation rates, pointing accuracy, etc.) is continuously updated. The simulator also takes into account the commands generated by the software to update the state of the spacecraft's actuators.

Each test case is run via a script that sets the initial conditions of the system, calls the simulation program, launches the ACS software, and finally collects and displays the results. To allow better control of the software, some additional routines were written. These routines allow the tester to start the ACS in a particular mode (after staying in mode 0 for a moment to initialize the filters), collect directly the parameters of interest for the test, and provide complete control of the paddles deployment sequence, the AUX errors, the time, etc. Using this setup, the test cases can be implemented easily and can be repeated as desired because all the data necessary for initialization and the complete information extraction process is stored in the script.

4.1 Blackbox Testing

Because the same types of tests must be run for every mode, the testing process used the same three-step process for blackbox testing of each mode:

Switching logic testing: In this step, the goal is to verify that the ACS enters and exits the modes when the parameters take on the expected values and when the mode delay has elapsed.

Parameter testing: The ACS software senses the satellite through a set of parameters, which are then exploited to decide to send some commands to the actuators or to switch mode. For the ACS to perform correctly, the integrity of these parameters is crucial; hence it must be verified that they really reflect the physical state of the satellite.

Functional testing: Finally, it is necessary to make sure that each mode accomplishes its fundamental task correctly. We need to check that the ACS interprets the parameters correctly and then issues the right commands that have the expected effect on the spacecraft's attitude.

4.2 Whitebox Testing

The whitebox testing process was divided into two different steps: coverage evaluation and the design of additional test cases.

The goal of coverage evaluation is to identify the parts of the code that have been left unexplored by blackbox testing and therefore could contain additional errors. Three different tools were used to help in assessing the level of coverage achieved by the blackbox test cases:

- Attol Test Coverage from Attol Testware [1]: This tool is compliant with Level A of DO-178B (every point of entry and exit in the program is checked using the MC/DC criterion).

- Cantata from IPL [3]: This tool is not fully compliant with level A of the standard because the C version of the tool uses the masking version of MC/DC, not the unique cause version.

- GCT (Generic Coverage Tool, Free Software Foundation): This tool only supports decision coverage or multiple condition coverage, not MC/DC, and it was not designed specifically for the DO-178B standard. However, it is the coverage evaluation tool used for the regression tests of the HETE software, so it was also included in the coverage evaluation process.

The coverage evaluation revealed that some parts of the code were not fully covered (according to the MC/DC criterion) by blackbox testing. Additional test cases were developed to fill the gaps.

5 Analysis of the Results

This section of the paper describes the blackbox and whitebox testing results as well as discussing the implications of the results on the relation between MC/DC coverage and software safety, the complexity of satisfying the MC/DC criterion, and the difficulty of achieving MC/DC coverage in this case study.

5.1 Blackbox Testing Results

As expected, the various types of errors found during blackbox testing were often associated with off-nominal test cases, particularly in the mode switching logic. For example, unwanted mode switching was found to occur when a variable *time-in-mode* took a negative value. Although this should never happen, it could result from a bad initialization. Because *time-in-mode* is declared as a long unsigned integer, it should never be able to take on negative values. However, somewhere in the code it is converted to a long signed integer. During testing we found that in every mode, an out of range *time-in-mode* value will bring the ACS into the next mode if the other parameters allow it even if the required mode delay time has not elapsed.

A second example of an error detected by blackbox testing was that the required delay in mode 3 was not taken into account for switching induced by one particular parameter although it is taken into account when switching is the result of a different parameter. An examination of the logic detected confusion in the *if-then-else* branching logic for mode 3 switching.

Other errors detected included missing default cases, an incorrect definition of a threshold (the value should have been 1.7453e-3 rad/s or 6 arcmin/s, but instead was set to 1.7453), missing conditions, and the lack of a required absolute value *abs()* function in some computations.

5.2 Whitebox Testing Results

The whitebox coverage analysis revealed some parts of the code were not fully covered (according to the MC/DC criterion) by blackbox testing. As might be expected, these parts primarily involved error-handling. This fact is consistent with data showing that a large percentage of operational errors found in requirements involve the error-handling routines, which are often not well tested. These routines are difficult to test during functional testing as they involve unexpected and erroneous behavior of the software, the environment, or the underlying digital hardware, such as bit flips caused by EMI. Some examples of the types of uncovered error-handling code for HETE-2:

- The handler that switches modes has a default that handles erroneous modes (i.e., modes outside 0–9), which was never exercised. Similar unexercised code involved checking for incorrect values (1) in the last *else* clause in a statement in which one of the preceding clauses will always be taken unless there is an error in other parts of the code or (2) in a short-circuited Boolean expression in which the first clause will always be true unless an incorrect path has been taken in the software to reach the decision statement. Another mode-related testing omission involved the situation where the satellite enters mode 6 with the paddles already deployed, in which case it should immediately return to mode 5.

- For redundancy, there are two magnetometers on the spacecraft. The detumble, spinup, and reorient control logic selects either magnetometer A or magnetometer B using an *if-else* statement. Because a failure never occurred during system testing, this decision statement is not covered. In another routine, a test is made for an erroneous magnetometer value but this test was never exercised during system test.

- The software checks for errors in the sensor data to detect possible errors in magnetometers A and B, in the different sun sensors, or in the wheel tachometer. If an error occurs, the data of the corresponding device is not updated and the old values are used. No AUX errors were simulated during blackbox testing, so the branches of the code that handle these situations were not executed.

Some limit cases were also never reached during system test. For example, when the satellite is not on station (i.e., for modes 0,1,2,3,4, and 6), the wheel torque is limited by a function called *control-wheel* to a maximum of the absolute value of 0.02 Nms. During system test, the negative torque limit was never reached so a decision in the *control-wheel* routine was left uncovered. In another routine, a function called *limit-mag-moment* limiting the value of the processed torques in the on-station controllers is never called. The physical limits for the coil torques is given by $V/R \times A_{eff}$ (where V is the bus voltage, R is the coil resistance, and A_{eff} is the effective area of the coil). In the simulations, the torques returned by the on-station controllers sometimes go well above this limit. However, they are later limited by another function that processes the raw commands and sends them to the actuators. So the magnetic moments of the coils are actually limited (albeit in another part of the software), and it was determined after the coverage evaluation that the function *limit-mag-moment* was not needed. Other instances involving limit checking in what was determined to be dead (unreachable) code were also detected and the code removed.

Another unexercised part of the code resulted from a change in the software that was not implemented everywhere in the code. The inertia matrix of the satellite is different if the paddles are deployed or not. The controller selected the correct inertia matrix using the following decision:

```
if (rom->paddles-deployed == 1)
      use I_deployed
else
      use I_stowed
```

The condition *rom->paddles-deployed* == *1* is a holdover from an old version of the code: initially the state of the paddles was a binary variable (0 for paddles stowed and 1 for paddles deployed) but it was decided later that the deployment of the paddles should be monitored individually for each paddle. Therefore, the state of the paddles was changed to be denoted by four bits (0x0 for all paddles stowed and 0xF for all four paddles deployed). In the version of the code that was tested, several places in the code were not updated and the single-bit notation was still used. This error results in a bad selection of the inertia matrix for the on-station controller when the paddles are actually deployed (*rom->paddles-deployed* is not equal to 1 when the paddles are deployed). The two inertia matrices are not very different, so the bad selection was not noticed in the blackbox simulations, and the error was revealed only by the whitebox testing.

The ACS software also watches for time rollovers, i.e., when the present time is smaller than the time of the previous sample. Time rollovers can occur when a time register reaches its maximal value (this should never happen on HETE-2 because a 64-bit digital clock permanently keeps track of the time) or after a reboot of the processor. Such timing glitches cause a problem for the ACS software, particularly for the calculations of the time derivatives and the wheel speed, which use the time differences between two samples. If a time rollover does occur, the software is supposed to use the old values for the magnetic field time derivative and the tachometer wheel speed instead of computing new ones. If the sensor data is too old by the time the commands to be sent to the actuators are computed, the software sets all the commands to zero, thus ensuring that no out-of-date commands are executed, for example after a processor lockout. These timing checks in the ACS software were never exercised during blackbox testing.

Other sanity checks were also found to be unexercised. An example is the verification of the bus voltage before the computation of the commands for the torque coils drivers. If the bus voltage reading is too low (less than one volt) or too high (more than 100 volts), the reading is assumed to be erroneous and a nominal voltage (28V) is used instead. Erroneous bus voltages did not occur during blackbox testing.

The MC/DC coverage evaluation also uncovered errors in the specification. An error in the code involving the *paddles-deployed* variable caused one branch of the mode switching logic, which goes from mode 2 to mode 8 when the cameras are tracking and the paddles are deployed, never to be taken. This branch was not included in the specifications, so the problem was not noticed during blackbox testing.

In the DO-178B specified process, after the parts of the code not covered by blackbox testing (with respect to a particular coverage criterion) are identfied, additional test cases must be designed to fill the gaps. Additional HETE-2 test cases were generated to test the detection and handling of illegal modes; to determine whether correct behavior occurred when the satellite enters mode 6 with the paddles already deployed (the ACS should switch immediately back to mode 5 and the on-station cycling behavior continue normally, which it was found to do after the additional tests were run); to determine how the ACS software handles AUX errors that can corrupt the data coming from the different sun sensors and from the wheel tachometer; to test the backup magnetometer selection logic; to fully cover the code that generates the torque coil commands; to test limit cases (threshold handling) in the torque coil and wheel torque; and to throughly test time rollovers and the use of obsolete sensor data.

Most of the new testing showed the software to be correct, but some previously undetected errors were found by the additional test cases generated to ensure MC/DC coverage. One such case involved handling AUX errors. When an AUX error is detected on only one of the sun sensors, the software does not try to use the other sensors to compute the sun-pointing parameters, but instead discards all the sun sensor information. This algorithm optimizes for short AUX bus blackouts, in which it is not worthwhile to lose time going through complex selection logic to pick up the good information since the data will be available again a few samples later. However, the logic needed to be changed to handle the case of a sun sensor hardware failure that produces a permanent error.

In another example, the selection of an incorrect magnetometer value in a test revealed an error in the code that handles this error (a pointer is never set and when it is referenced later in the program, it causes a segmentation fault).

5.3 Relation Between the MC/DC Criterion and Software Safety

A main question in this study was to determine whether MC/DC coverage improved the safety of the software. In other words, did the additional tests required by the coverage criterion find important errors or did they just consist of playing with some variables to artificially toggle conditions, resulting in a process in no way related to safety or even to practical issues in software engineering.

We found that for the HETE-2 software, all the additional tests required to satisfy the MC/DC criterion were directly linked to an important feature of the software. More precisely, the need for additional tests corresponded to four kinds of limitations of the blackbox testing process:

- Something was forgotten during blackbox testing, for example, the case where the sattellite enters mode 6 with the solar paddles already deployed.
- The software has a complex logic mechanism requiring in-depth understanding and precise, customized testing. This was the case for the magnetometer selection logic. Much of the untested code was involved in error-handling.
- Some feature of the software was not included in the specification and therefore could not give rise to a test case in a blackbox testing context. This was the situation for the AUX error checks and the time checks, and for the bus voltage verification before the coil torque computations. The fact that the whitebox testing process served as a verification of the completeness of the specification was very useful.
- The effects of some errors were too small to be detected by blackbox testing. In the case of the *paddles-deployed* variable, we found that a bad value was assigned to this variable because the conditions in which it was involved could not be toggled. The consequences of this error had gone undetected previously because the difference in output was so small.

5.4 Complexity of Satisfying the MC/DC Criterion

A second question to be explored was whether MC/DC coverage was excessive. That is, given that whitebox testing is important in ensuring safety, would it be possible to use a lower level of coverage and still ensure the same level of safety?

The answer to this question for the HETE-2 software was clearly no. For example, an important problem was detected only because the MC/DC criterion required checking the second condition in an conditional expression. Test cases could have satisfied decision coverage and condition/decision coverage without uncovering this problem. As it turns out, the problem was not crucial—it would not have caused any damage had it remained undetected. However, the test cases that detected an error concerning a critical system variable, *paddles-deployed*, involved the same kind of Boolean function except that the AND operator was replaced by an OR. The fact that this important problem could also be detected by decision coverage relies only on this small difference and in general would not be true.

5.5 Difficulty of Achieving MC/DC Coverage

A final question concerns relative cost. Two comparisons were made.

First, we calculated the time required for whitebox testing in comparison to the total test time. In the case of the HETE-2 software, the coverage determination and the design of the additional test cases represented about 40% of the total testing time (the rest was devoted to blackbox testing). Note that powerful tools were used to help determine coverage, so the coverage evaluation process was quite fast and easily repeatable.

One feature of the software, inherent in the nature of the code itself, facilitated whitebox testing: part of the blackbox testing activity consisted of checking the mode switching logic, i.e., verifying that the different branches of the switching diagram were taken under the correct conditions. These switching specifications are, in fact, very close to the structure of the source code itself, so part of the blackbox testing was equivalent to testing the source code structure. Therefore, the number of required whitebox tests was reduced, and the proportion of whitebox to blackbox testing time was biased in favor of whitebox testing. Therefore, white box testing was useful in finding errors in the code, but it indeed represented a time-consuming step in the complete testing process.

Second, it is interesting to compare the difficulty of achieving MC/DC coverage versus achieving a simpler form of coverage such as decision coverage. In fact, we found in the case of HETE-2 that MC/DC coverage was not much more difficult to achieve than decision coverage. This result was due to the programming style of this code: only 11% of the decisions were composed of Boolean functions while all the other decisions were single-condition decisions. In this latter case, decision coverage and MC/DC are equivalent.

An important tradeoff is involved here. The fact that decisions were kept simple contributed to making this software well suited for MC/DC testing. However, keeping the decisions simple in the source code also has its drawbacks—in general simpler decisions lead to more complex logical structure. In essence, multiple conditions linked by Boolean operators were replaced by nested *if-else-if* instructions. This kind of logic is very prone to errors (as demonstrated, for example, in the switching logic testing of mode 3). So although this style of coding facilitates MC/DC testing, it may also lead to more errors in the code and leads to code that is also more difficult to read and maintain.

6 Conclusions

Although this case study provides only one instance of an evaluation of the MC/DC coverage criterion, it does provide examples of its usefulness and effectiveness. Functional testing augmented with test cases to extend coverage to satisfy the MC/DC criterion while relatively expensive, was not significantly more expensive than achieving lower levels of code coverage. Important errors were found by the additional test cases required to achieve MC/DC coverage (i.e., in the software found not to be covered by blackbox functional testing). The use of automated tools to evaluate coverage was helpful in reducing the costs of structural coverage testing.

References

[1] Anon. *Attol Coverage Documentation*. Attol Testware, http://www.attol-testware.com.

[2] Anon. *HETE Documentation*. MIT Center for Space Research.

[3] Anon. *Cantata for C Documentation* , IPL, http://www.iplbath.com/p13.htm.

[4] Anon. *Software Considerations in Airborne Systems and Equipment Certification*, DO-178B RTCA, Washington D.C. 1992

[5] Chilensky, J.J. and Miller, S.P. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 1994.

[6] Jasper, R., Brennan, M., Williamson, K., Currier, B., and Zimmerman, D. Test data generation and feasible path analysis. *SIGSOFT Software Engineering Notes*, ACM, 1994.

[7] Simonetti, A., Coupier, A., and Secher, C. Experience gained from recent avionics software certification. *Bulletin techniques du Bureau Veritas*, 1992.

[8] Myers, G.J. *The Art of Software Testing*. John Wiley, 1986.