Defence Science Journal, Vol 49, No 4, August 1999, pp. 317-322
©1999, DESIDOC

Tools and Techniques for Testing of Flight-Critical Software

G. Venkat Reddy and Ahalya Chandrasekhar
Aeronautical Development Establishment, Bangalore-560 093.

ABSTRACT

Flight control system software is a critical component of the digital flight control computer of light combat aircraft. The problems associated with the testing of flight critical software and the test tools, and techniques used to achieve maintainability, and structural and functional coverage of test cases are presented. Also, the experience gained throughout the cycle of testing-design and implementation, reviews and revisions, test execution and software error detection and modificaton of test cases based on requirements and design changes, and regression testing are enumerated. It presents an object-oriented approach towards testing to make it less tedious, more creative, reviewable and easily maintainable.

1. INTRODUCTION

Software testing is an important aspect in the software development lifecycle. Only its role and expectations are controversial. While the literature survey on software testing indicates that the testing can only claim for the errors found, and the project management claims that the software has matured and become reliable after testing. Whichever way one looks at it, software testing builds confidence in the software product and cuts down the cost and time for software repair during the system level testing if most of the errors are found during the module level and software integration level testing. The test results are subject to the scrutiny of the certification agency for the eventual acceptance of the product.

The requirement of software testing are stringent for flight-critical software. A process is worked out, based on DOD-STD-2167A, for carrying out the task in an efficient manner. This paper describes the process, the techniques and the tools used during the testing of flight control software of light combat aircraft (LCA) developed at the Aeronautical Development Establishment (ADE), Bangalore. It also presents a case study of an Aeronautical Development Agency (ADA) package of LCA-FCS software–PACT–MNGR, which deals with primary actuators.

2. SOFTWARE TESTING PROCESS

Testing is a continuous activity right from the start of the lifecycle to the operational use of the product. The software gets tested everytime it gets executed. However, a systematic process is evolved to find out errors at the earlier stages, so that the manifestation of errors/faults should not result in the failure of the system at a later stage which will cost the project more.

The guiding principle of quality–PDCA (plan, do, check and act) cycle is followed. First, a test plan is developed which forms the guiding document for the testers on how to proceed with the testing, what are the minimum requirements of testing which shall be adhered to, like test coverage criteria, etc. This test plan is a deliverable item and is formally reviewed.

Received 03 June 1999

The following steps have been described in the test plan: (i) review of design and code by testers, (ii) generation of test traceability matrix, (iii) guidelines for generating test cases, (iv) test drivers, (v) test procedures, (vi) test execution, and (vii) test report generation. A separate test team, independent of design team, is established to carry out testing.

The first activity which the tester should undertake is to review the design and code for testability and traceability of requirement to design. A thorough design and code walkthrough should be carried out and also non-execution-based testing should be carried out for different test scenarios wrt requirement. This has enormous benefits, and the experience shows that 60–80 per cent errors are detected during these reviews. The cleanroom engineering methodology that is getting vague also emphasises the advantages of pre-execution testing activities as experienced. The checklists are filled to this effect.

The software test description (STD) document is prepared, while the detailed coding is carried out. STD consists of test cases, i.e., the test input and the expected output. These are reviewed during test readiness review (TRR) activity. STD should be well-documented, and the purpose of each test case should be elaborate enough for maintainability, reviewability for correctness and completeness. The test cases are generally documented in input and expected data files which form the part of STD.

The test drivers and stubs are developed for module testing. The test driver reads the input from the input data file and assigns these values to the right data structures for input as documented in the input file and calls the unit under test (UUT) module and acquires the values from the different data structures for output and writes these values to an actual output file in the same order of output as documented in the expected output file.

The actual output and the expected output are compared using a comparator tool. The discrepancies are analysed and the code is debugged to validate the discrepancies. A software problem report is generated to track a fault and its corrections. After a test execution is complete, the test cases are subjected to coverage analysis to verify whether the path coverage or decision-to-decision coverage is adequate. The test cases are augmented if found inadequate.

The test procedures are developed to automate the process of test execution, i.e., compile, link, run and compare expected and actual output and also to do coverage analysis. A test report is generated documenting the module being tested, the revision number of module, the date of testing, the status of testing pass/fail the coverage and the remarks.

3. TESTING TECHNIQUES

The philosophy of the testing techniques to be adopted to generate test cases is documented in the test plan. The two basic approaches are: black box testing, (i) and (ii) the white box testing techniques. First, modules are tested in stand-alone fashion and then gradually integrated in bottom-up fashion, the main emphasis in the integration testing is on the testing of interfaces.

3.1 Black Box Testing

Black box testing, also known as functional testing, or specification-based testing, assumes the module being tested as a black box whose internal details are not known. It is like testing a custom-built integrated circuit with the knowledge of its functional specifications and input-output pin details and without knowing its internal circuitry. If every possible value for each input and all possible combinations of input are taken into account, the number of test cases become so large that it will be humanly impossible to execute these with all available resources. Hence, some techniques have to be evolved which will help the tester to generate reasonable number of finite test cases like equivalence partitioning, boundary value analysis, cause-effect graphing and error guessing, etc.

3.1.1 Equivalence Partitioning

The word equivalence partitioning is derived from the modern algebra's set theory. In software testing parlance, it means partitioning of input space, though not strictly in mathematical sense, i.e., if one value for the representative class (partition) is tested, it can be assumed that the test is valid for all values of that representative class. Unlike equivalence classes of

mathematical sense, these sub-divisions of input space can be overlapped.

3.1.2 Boundary Value Analysis

The boundary value analysis is based on the hypothesis that we are likely to commit mistakes at the boundaries of linear regions. An analogy can be that accidents are more likely to happen at the turnings or cross-roads than on the straight roads. Test cases have higher yield of fault detection at the boundary values of input equivalence classes than any representative values.

3.1.3 Cause-Effect Graphing

The program is visualised as a transformation of input conditions or states (causes) to output conditions or states (effects). A graph can be drawn linking the causes and effects with the relations like AND, OR, NOT. Equivalently, a decision table or a set of Boolean expressions can be derived from the specifications. This will give an insight into the requirements as well as provide the test cases which need to be exercised to test the functionality of the software.

3.1.4 Error Guessing

Some test cases can be derived based on intuition or experience. There are no set rules for error guessing but individual experiences of the test team members can be shared among all the other members about the kind of test cases that have likely high yield of fault detection, may be typical to the domain-dependent or processor-dependent.

3.2 White Box Testing

White box testing, also known as structural testing or code-based testing, assumes the complete knowledge of implementation details of the module being tested. The idea of this testing technique is to test all the statements and paths inside the code.

Statement coverage implies that all the statements are executed during testing. Assignment expressions of the nature, output = a1*inp1 + a2*inp2 + a3*inp3 can be tested by setting one input at a time and monitoring the output. Assignment of Boolean expressions (using AND, OR and NOT) can be tested with $< 2^n$ test cases. Simple Boolean expressions like bool–outp = bool–inp1 && bool–inp2 && bool–inp3 can be tested with 4 test cases TTT, TTF, TFT, FTT. If there are complex Boolean expressions having exclusive-OR etc. then exhaustive testing of the expression with 2^n test cases will be required.

Path coverage implies that all the path combinations are executed during testing. If all the possible paths are taken into account, it will be a combinatorial explosion which will be humanly impossible to execute and test with all the available resources. Hence, some techniques like decision coverage, decision/condition coverage, loop coverage, etc. have been evolved.

When the test cases are designed using black box methodology based on functionality, it is observed that they cater for more than 90 per cent of the structural coverage. Whichever structural coverage is missing, that can be appended by additional test cases.

3.2.1 Decision Coverage

Each decision in the module should be executed for its TRUE and FALSE values, such that if-then-else paths are exercised fully. The number of decision paths in the module is called Cyclomatic complexity number, denoted by VG, which is equal to E-V+1, where E is the number of edges, and V is the number of vertices in the graph representing the module. This gives the lowest upper bound for the number of test cases for path coverage. There are many tools (static analysis) that give VG, while other test tools (dynamic analysis) give the decision coverage for the test cases.

3.2.2 Decision/Condition Coverage

If the decision consists of a condition or a Boolean expression of multiple conditions and Boolean variables, then the test cases should be developed to give the coverage of conditions and Boolean expressions. A condition like $x <= 2$ requires a minimum of 3 test cases. When multiple conditions exist in the expression, judicious selection of test cases should ensure that each condition is independently tested, while other conditions are set to proper default values. There are no tools that can verify the decision/condition coverage of a given set of test cases.

3.2.3 Loop Testing

The loops should be tested such that the termination of a loop is proper wrt terminating condition (decision/condition coverage) and by executing the loop for nil times, one time, typical number of times, n-1, n and n+1 times (when loop parameter is compared against n count) as it is possible to test.

3.3 Module Integration Testing

After the modules are tested independently for their specified functionality and structural coverage, the integration of modules is tested. The testing is done to verify that the definition of interfaces is consistent across the modules and the modules together satisfy the requirements. This approach is bottom-up integration method. If required, the dynamic execution of modules is done to verify the functionality of dynamic elements like filters, transient free-switches, persistency checks.

3.4 Hardware/Integration Testing

After the testing of software modules (stand-alone and integrated) the software is tested end-to-end in embedded hardware in a simulated test environment (Engineering Test Station). The hardware input are set and hardware output and some software output are monitored to verify and validate the requirements.

4. TESTING TOOLS

Various tools that are used in the testing process should enable the testing easier, efficient, less tedious and more creative. Some of the tools that are used in the testing are mentioned below.

4.1 Software Simulator

Software simulator is a tool that simulates the processor's execution environment, i.e., the instruction set, call mechanism, registers, interrupt and fault handling mechanisms, floating point arithmetic, etc. Hence, it allows the cross-compiled code to be executed and tested on the host environment well before the target system development is complete. It also helps in debugging the software, both at the source level and the machine level. It may not be feasible to test the module level functionality for some of the modules end-to-end where there may be strobing or setting and resetting of the signals. The only way to test and document the testing is by using the debugger of the simulator.

4.2 Static & Dynamic Analysis Tools

There are software tools that assess the path coverage and code complexity during testing. *Logiscope* and LDRA testbed tools are some of the popular tools. Logiscope evaluates the coding and design quality through static analysis of source code. The dynamic analysis of code is used to measure path coverage.

4.2.1 Static Analysis Tool

ADA static Analyser tool takes ADA source code as input and provides the various measurements of the code like number of statements, cyclomatic complexity and many other parameters that provide quality metrics for the design.

4.2.2 Dynamic Analysis Tool

The dynamic analyser tool instruments the source code with execution trace call procedures. When the instrumented code is executed with the test cases, the trace calls will generate traces in the raw execution results file. It is these traces that allow the path coverage to be evaluated.

4.3 Comparator Tool

This in-house developed tool is used for comparing the expected and the actual output. It generates a report containing the number of total test cases, the number of test cases passed and the number of mismatches along with the actual and the expected output values with discrepancies and pass/fail status.

4.4 Report Generator Tool

This in-house developed tool automates the generation of the test report by compiling information from different test output files.

5. OBJECT-ORIENTED TEST CASE GENERATOR TOOLS

There are several tools for autocode generation which are quite good for well-defined applications. The feasibility of autotest case generation was studied for

testing the implementation of control law software. The control law specification depicts the requirements in block diagrammatic fashion. This tool also visualises the requirements as an interconnection of well-defined objects (blocks). Each block's functionality is tested for a predetermined input-output specification: the interconnecting objects cooperate to set the input of the tested block as required. This tool has been developed in C++ using object-oriented techniques and verified for different requirements which could be depicted as an interconnection of well-defined objects. The development of the user friendly GUI is being carried out.

6. PRIMARY ACTUATORS SOFTWARE—CASE STUDY

The PACT–MNGR package, an ADA package of LCA-FCS software, is a very critical component in the flight control of the aircraft. It is responsible for driving the actuators, failure monitoring and failure detection of servo-electronics hardware and actuators, redundancy management and reconfiguration. There are 20 units in this package. A bottom-up testing is carried out to test this package. Test cases were generated for unit testing both from the functional and the structural point of view. The integration testing was carried out to test the interfaces and requirements.

The types of errors encountered are (i) requirements to design translation, (ii) design-to-code translation, (iii) cross-compilation of ADA code to *i*960 machine code, and (iv) validity of machine code in DFCC environment. Except the last one, all the remaining errors could be detected during software testing. The test cases and other relevant documents were reviewed by an independent review team which has verified it for correctness, test coverage and quality. All the suggestions of the review team have also been incorporated.

7. CONCLUSION

This paper presents the experiences gained during fight control system (FCS) software testing. In LCA-FCS project, the testing is carried out at various levels like software testing, system integration testing in minibird environment, validation of FCS in ironbird environment, aircraft testing on ground and finally in air. It was observed that there may be some changes in the requirements based on the feedback from the tests at different levels due to changes in the values of different parameters. It is necessary to incorporate changes in the design and carry out testing in the shortest possible time. Tools and techniques used to carry out testing play an important role in achieving this. The test case generator itself will be helpful in quickly modifying the test cases. It is hoped that the special tools developed for this purpose will be useful in the future work on LCA-FCS.

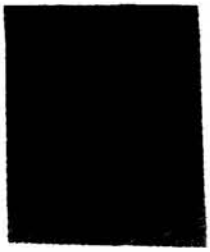
REFERENCES

1. Myers, Glenford J. The art of software testing. John Wiley & Sons, New York, 1979.

2. Beizer, Boris. Software testing techniques. Ed.2. Von-Nostrand, New York, 1990.

3. CSU/CSC test plan for primary OFP of LCA-FCS. Aeronautical Development Establishment, Bangalore. Report No ADE/LCS/FCS/PRY/STP, January 1996.

Contributors



Mr G Venkat Reddy Joined Aeronautical Development Establishment (ADE), Bangalore, in 1987. He is working in the area of flight control sytems for the last 12 years. He is a member of the test team carrying out software and system integration testing of LCA-FCS.

Smt Ahalya Chandrasekhar joined Aeronautical Development Agency (ADA), Bangalore, in 1986. She is working in the area of flight control systems for the last 13 years. Recently, she has been deputed to ADE, Bangalore, where she is the team leader for the CSU/CSC testing of LCA-FCS software. She is also involved in system integration testing of LCA-FCS.