GOAL-ORIENTED AUTOMATIC TEST CASE GENERATORS FOR MC/DC COMPLIANCY

Emine G. Aydal, Jim Woodcock and Ana Cavalcanti
University of York, UK

Keywords: Test case generators, goal-oriented test-case generators, MC/DC.

Abstract: Testing is a crucial phase of the software development process. Certification standards such as DO-178B impose certain steps to be accomplished in testing phase and certain testing coverage criteria to be met in order to certify a software as Level-A software. Modified Condition/Decision Coverage, listed as one of these requirements in DO-178B, is one of the most difficult targets to achieve for testers and software developers. This paper presents the state-of-the-art goal-oriented automatic test case generators and evaluates them in the context of MC/DC satisfaction. It also aims to guide the production of MC/DC-compliant test case generators by pointing out the strengths and weaknesses of the current tools and by highlighting the further expectations.

1 INTRODUCTION

Civil Aviation is only one of the fields where the proper use and the correct implementation of software are of high importance in order to protect the lives of the passengers. The Federal Aviation Administration (FAA) is the body in the United States of America (USA) that "*is primarily responsible for the advancement, safety and regulation of civil aviation as well as overseeing the development of the air traffic"*(Hayhurst et al. 2001). In order to secure FAA approval of digital airborne computer software, the developers are recommended to use the RTCA/DO-178B document (DO178B 1992), by the FAA through Advisory Circular (AC) 20-115B (DO178B 1992, Hayhurst et al. 2001, AC#20-115B 2003). In RTCA/DO-178B, software life cycle activities and design considerations are described and sets of objectives for the software life cycle processes are enumerated. According to RTCA/DO-178B document, one of the most difficult objectives to be met in order to achieve Level-A software is the satisfaction of the Modified Condition Decision Coverage (MC/DC). MC/DC is a test coverage criterion that verifies the adequacy of the executed tests in terms of conditions, decisions and their relations with respect to each other. The process of satisfying MC/DC for software is still computationally complex and therefore software developers are in urgent need of a tool that automates the process of generating test cases that cover MC/DC or a tool that is able to verify that a given test suite satisfies MC/DC.

Within this context, this paper focuses on the goal-oriented test case generators. We commence with a brief overview of MC/DC and its positioning in the literature. We, then, summarize the general concept of test case generation by explaining the principal components of generators in an organised fashion. We present the capabilities and weaknesses of various goal-oriented test-case generators and conclude with a summary of practical improvements that would ease MC/DC satisfaction when implemented.

2 MC/DC IN THE LITERATURE

One of the main difficulties encountered in the testing phase is the decision of adequacy (decision of terminating the testing process). Testing can continue as long as there are different, untested execution paths and/or requirements that have not been verified, but the constraints of software development, such as time and budget limitations, only allow a certain amount of test cases to be carried out. Therefore, different sets of rules that prescribe some property of the test sets are suggested in the literature (Kapoor and Bowen 2004, Ammann et al. 2003, Chilenski and Miller 1994). These sets of rules are named as *test coverage criteria*. The satisfaction of a test coverage criterion

G. Aydal E., Woodcock J. and Cavalcanti A. (2007).
GOAL-ORIENTED AUTOMATIC TEST CASE GENERATORS FOR MC/DC COMPLIANCY.
In *Proceedings of the Second International Conference on Software and Data Technologies - SE*, pages 290-295
DOI: 10.5220/0001324002900295

that verifies the adequacy of the executed tests in terms of desired qualities is checked during the test coverage analysis phase. The two different test coverage analyses (Hayhurst et al. 2001, Adrion et al. 1982) are requirements coverage analysis and structural coverage analysis. Requirements coverage analysis forms a bridge between software requirements and test cases, thus demonstrating how well testing has verified the implementation of the software specifications. Structural coverage analysis provides traceability between code structure and test cases. The results of this analysis shows how much of the code structure has been executed.

Whilst determining the percentage of the code structure covered, different structural coverage criteria may focus on different elements in the program and their accuracy, effectiveness and cost may vary. In (Chilenski and Miller 1994, Chilenski and Newcomb 1994), structural coverage criteria are investigated within three categories:

- **Data flow coverage criteria** deal with the interrelationships along subprogram subpaths between points where a variable is defined and where that variable's definition is used.
- **Control flow coverage criteria** analyze the interrelationships of decisions and conditions along subprogram path subsets.
- **Control coverage criteria** check the program by examining decision and condition outcomes and interrelationships within a single control point.

Modified Condition / Decision Coverage (MC/DC), being a control coverage criterion, deals with decisions and conditions in control points. A **condition** is a leaf level Boolean expression, which does not include any Boolean operators and thus cannot be broken down into smaller Boolean expressions. A **decision** is a Boolean expression that is composed of a single condition or expressions that combine many conditions (Hayhurst et al. 2001, CAST 2001). MC/DC requires the following statements to be true for the given set of test cases (Hayhurst et al. 2001, Kapoor and Bowen 2004, Ammann et al. 2003):

- Every point of entry and exit in the program has been invoked at least once.
- Every decision in the program has taken all possible outcomes at least once.
- Every condition in the program has taken all possible outcomes at least once.
- Every condition in a decision has been shown to independently affect that decision's outcome

3 TEST CASE GENERATION

Testing has been one of the major processes in the software development accounting for approximately 50% of the time and over 50% of the budget (Chilenski and Newcomb 1994). Due to the size of the input space and the number of paths in a program, the possibility of being able to complete exhaustive testing remains low and instead, researchers have been looking for some other ways of quickening the testing process with the aim of achieving some testing coverage criteria. Automation of test case generation with the help of a tool is one of the approaches aiming to reduce the time and effort given to testing.

This chapter will first explain the concept of automating the test case generation process in general and section 3.2 will provide more insight to the components expected in test case generators.

3.1 Automatic Test Case Generation

The automation of test case generation may have different targets to achieve. The test case generation for functional testing aims at generating test cases that exercises all the functions of the systems either based on specifications or based on the model of the system. The test case generation for structural testing, on the other hand, attempts to find a set of program inputs X that achieves desired testing coverage criterion, provided that X is a subset of the set of all possible input combinations (Tracey et al. 1998). This type of test case generation may be *code-based*, *model-based* or *specification-based* depending on the approach taken.

In fact, the approach taken not only determines how the test cases will be generated, but also whether the system has to be executed in the generation of these test cases. Some test case generation techniques need the execution of the program with the generated test suite. These techniques are called to be *dynamic*. If the execution of the system is not involved in the generation of the test suite, then these techniques are said to be *static* techniques.

The next section will briefly explain the components that are likely to be seen in test case generators.

3.2 General Structure of the Test Case Generators

A typical test case generator consists of three parts; *Program Analyzer, Strategy Handler* and *Generator*.

Figure-1 shows the components and the tasks that may be handled in these components.

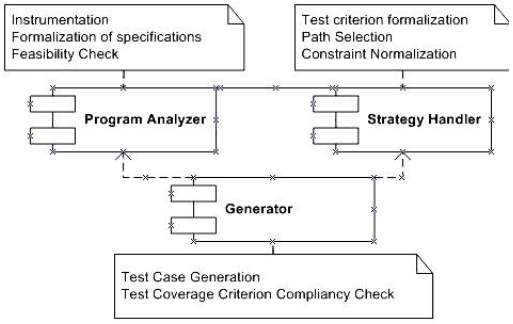

Figure 1: The components of a test case generator.

The first component, *Program Analyzer*, is responsible for the preparation of the program to the automation process. This preparation may need some changes on the actual program or some analysis of a certain property. For instance, some test case generation techniques require specific response from the software and therefore the instrumentation may become necessary. If the approach taken is formal, a formal specification of the system may be in need. Another task performed at this point may be to build a control flow graph (CFG) or to perform parameterised feasibility analysis in order to locate as much unreachable code as possible.

Strategy Handler is the component where external factors affecting the test case generation are determined. This step may include the formalization of the test coverage criterion, the selection of paths, normalization of constraints, etc. In some cases, this component may need some amount of interaction or input from *Program Analyzer*.

Generator is the component that takes input from the other two components and generates test cases according to the rules of the approach followed. The satisfaction of test coverage criterion is generally verified in this component as well.

The model above is similar to that of (Edvardson 1999), however, the tasks of these components are more generalized in order to give reader an overall picture of the test case generators and establish a basis for the next chapter where different goal-oriented test case generators are discussed.

4 GOAL-ORIENTED TEST CASE GENERATORS AND MC/DC

There have been many studies on the automation of test case generation during the last two decades. Literature surveys on this field (Edvardson 1999, Prasanna et al. 2005) generally classified the test generators for structural testing into three categories; random, path-oriented, goal oriented. This paper mainly focuses on goal oriented generators. The reader may find further information on random and path-oriented test case generators in (Tracey et al. 1998, Durrieu et al. 2004, Diaz et al. 2004).

The goal-oriented generators generally identify test cases covering a selected goal such as statement or branch coverage irrespective of the path taken. In other words, in this sort of generators, the input generated does not necessarily traverse from entry to the exit point of the program, but may take an unspecified path. Since the paths are not restricted, the risk of encountering infeasible paths is reduced (Edvardson 1999). Having said that, the strategy selected should still provide a way to direct the search for input values.

Chaining Approach

One of the approaches followed is the chaining approach (Ferguson and Korel 1996). In this approach, for each test coverage criterion, different *initial event sequence* is defined and the *goal* nodes are determined accordingly. For instance, the initial event sequence for the branch coverage of the branch (p,q), the initial event sequence E is defined as $E = \langle(s, \emptyset),(p, \emptyset),(q, \emptyset)\rangle$ (Ferguson and Korel 1996). The problem with this approach is the lengthiness of the processes to prepare the program to testing. The *Branch Classification* process handled in the Program Analyzer has to determine the critical, semi-critical and non-critical nodes for each branch. Then, this classification leads the search during the execution of the program and decides which branch to take to reach the goal node or to cover the requested branch. Then again, for branch coverage, each branch has to be given explicitly to the generator and the program has to be executed as many times as the number of branches in the code. It is written in the specifications of the tool that the technique can be adapted to other criteria as well, but the algorithm to introduce a new criterion is not straightforward and the burden to introduce each unit of the criterion –eg. each branch for branch coverage - is still on the shoulders of the tester. The structure of the test case generator that uses *chaining approach* is given in Figure-2.

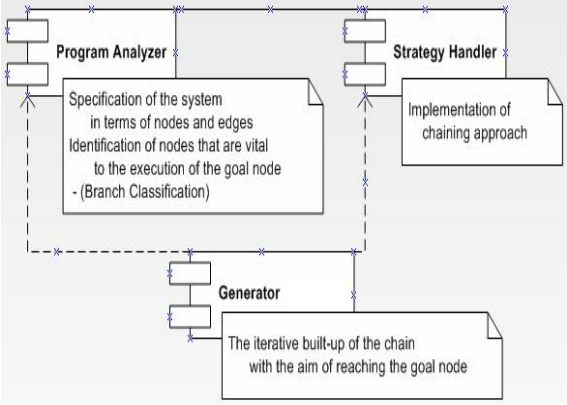

Figure 2: Chaining approach.

Simulated Annealing

A similar approach is studied in the domain of Heuristic-global optimization techniques (Tracey et al. 1998). Clark et al. have used *simulated annealing* in order to build a general framework for generating test-data. Figure-3 shows the internal structure of the generator described in (Tracey et al. 1998).

Given the representation of the candidate solutions and a *cost function* which can measure the quality of the candidate solution for a chosen test criterion, simulated annealing can overcome the problems of locally optimal solutions by performing modified neighbourhood search (Tracey et al. 1998). Instead of classifying branches to find the target branch, as in *chaining approach*, it uses the directions of the cost function. The cost function returns zero as long as the correct branch is taken along the path to be traversed.

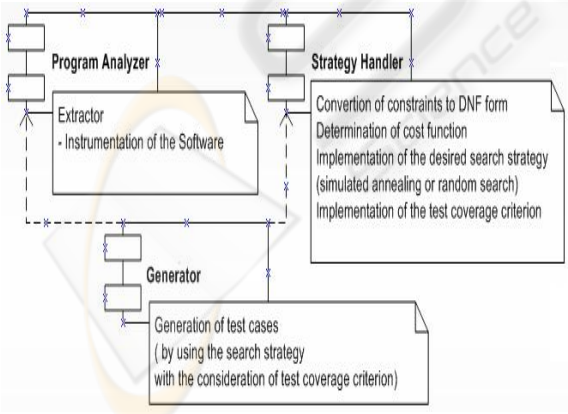

Figure 3: Simulated annealing.

This approach is superior to many others in the sense that it is flexible and allows changes during the procedure. The cost function plays the role of the oracle in finding the right path and avoids redundant branching. The test coverage criterion formalization is also hidden in the cost function. Thus, to generate test data for new test criterion, it is necessary to devise an appropriate cost function. The problem is that the cost function itself may need an oracle. Because finding a cost function for a new test criterion is not an easy process and to the best of our knowledge, no cost function has been produced to cover MC/DC. Furthermore, although the cost function aims to lead to the desired branches, there must be an additional unit to check that the generated test cases caused the desired coverage.

Having said that, the flexibility of the technique may allow further improvements. For instance, the derivation of the cost function from the test criterion may be formally demonstrated, and the degree of learning process in each execution may be increased possibly by harnessing the current optimization techniques such as tabu-search, generic-algorithms as well as simulated annealing with the help of software metrics (Tracey et al. 1998). These improvements may let researchers to adapt new criteria such as MC/DC by using more formal techniques.

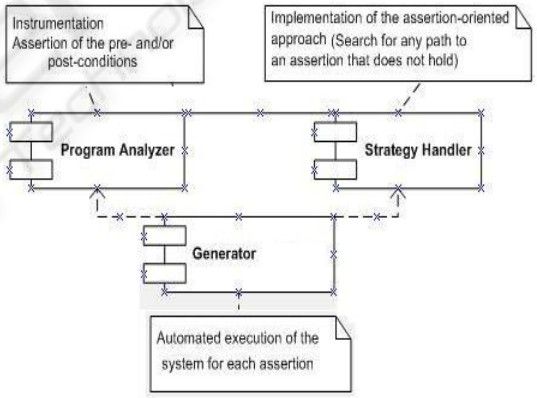

Figure 4: Assertion-oriented approach.

Assertion-oriented Approach

Another approach again in the context of goal-oriented generators is the *assertion-oriented approach.* This approach is based on the power of assertions, i.e. pre- and post-conditions. The approach may be useful when considered in the context of all-purpose test case generation. Because its main *goal* is to identify a program test on which an assertion is violated (Korel and Al-Yami 1996) and ultimately automate the process for each assertion in the program. By finding an assertion violation, it aims at finding a fault in the program, a

faulty precondition or an erroneous assertion. The Figure-4 shows its internal components.
For fault-detection purposes, this testing technique can be of use, but its drawback is that it does not consider any test coverage criterion within the process. Thus, there is no fixed termination point other than the number of assertions in the program, but as stated in (Korel and Al-Yami 1996), the problem of finding a program input on which an assertion is violated is undecidable and therefore the process may not be able to find violations for certain assertions and may never terminate.

ADA Testing Workbench

As a more formal approach in building test case generators, Chilenski and Newcomb (1994) examine *Ada Testing Workbench (ATW)*, a research tool that automates the analysis of a subset of Ada language and generation of coverage compliant specifications for twenty-one control, control-flow and data-flow coverage criteria, including MC/DC. The internal structure of ATW is shown in Figure-5.

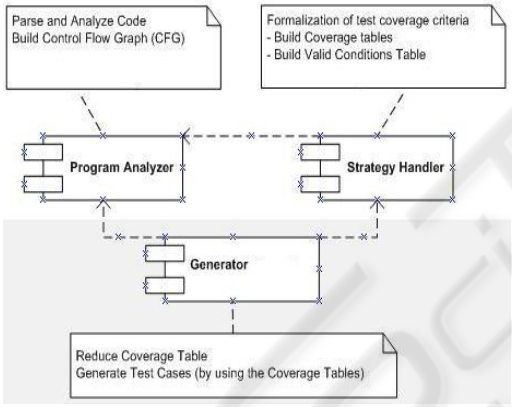

Figure 5: Goal-oriented test case generator – ATW.

The generator starts the process with the transformation of source code into a machine understandable representation. Then, ATW starts the parsing step where a set of knowledge base structures are generated. These constitute an abstract syntax tree (AST). By using AST, Control Flow Analysis generates CFG for each Ada subprogram. CFG is used to determine the decisions, conditions and variables that control the execution of the code. Meanwhile, coverage specifications are built for the test coverage criteria by using the coverage tables and feasibility analysis is carried out. Finally, the generator reduces the number of rows in the Coverage Tables by using Valid Conditions Table (VCT) and constructs test cases accordingly. Thus the *goal* of the generator can be summarized as generating test cases for a certain coverage criterion through Coverage Tables in an efficient manner.
Although this work was promising for test case generation, the efforts to improve the tool (ATW) ceased in 1994. One of the main benefits of this study has been that it demonstrated automated formal semantics capture techniques with derivation of formal specifications from Ada code and mechanical theorem proofs for properties of program paths (Chilenski and Newcomb 1994, Chilenski and Miller 1994).

Model-based Approach

In addition to the techniques and tools introduced, there are also model-based goal-oriented test case generation tools (Prasanna et al. 2005, Cavarra et al. 2002), most of which are based on UML. The test case generation from UML models is certainly possible. However, to generate test cases that satisfy a coverage criterion, the models need to be verified against the specifications. Although there have been some studies focused on formal verification of UML models, the overall semantics of UML still needs to be elaborated in order to get rid of ambiguities in its definition as stated by Prasanna et al. 2005 and therefore these tools are not studied in this paper.

5 CONCLUSION

Having evaluated a significant number of goal-oriented test case generators in the context of MC/DC satisfaction, we can summarize the crucial outcomes as follows:

- MC/DC must be formulated in a standard format and there must be a mechanism that introduces the criterion to the generator.
- If there is a need for the determination of all branches or paths in order to execute them, then this must be handled by the tool.
- Heuristic-global optimization techniques can be used if formality in the definition of cost function for MC/DC can be achieved.
- Analysis of the program code is an indispensable part of the test case generation since, like most other criteria, MC/DC is a syntax-dependent criterion. Some of the generators analyze the code through instrumentation, others through parsing and producing CFGs, etc. A formal analysis of the code that does not rely on the programming language can improve the confidence in the tool and provide more flexibility to other components of the generator.
- It is possible to produce tools that accept different coverage criteria in the form of plug-ins

as in (Chilenski and Newcomb 1994). Although our aim in this paper is to generate MC/DC-compliant test suites, having this broader idea in mind would probably facilitate the introduction of other criteria if need be.

- The components of the tool must comply with the 'separation of concerns' rule. *Separation of concerns* (SoC) is the process of breaking a program into distinct features that overlap in functionality as little as possible (Jackson 2006). In this case, the components; program analyzer, strategy handler and generator must be implemented in such a way that amendments in one of the components should not affect others in great deal. For example, if the instrumentation of the code in the program analyzer depends on the criterion formulated in Strategy Handler, this may cause huge amount of modifications when the decision for the criterion used is changed.

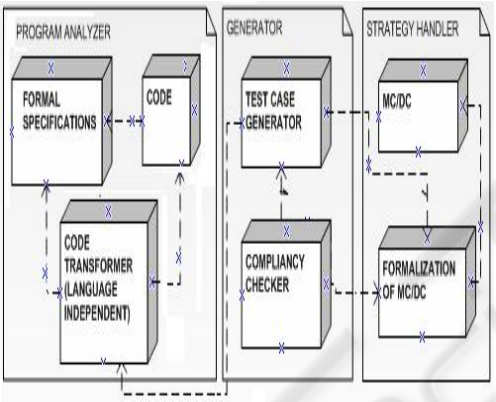

Figure 6: A different view to test case generation.

Figure-6 gives a different view to the test case generation process. The main idea emphasized in this figure is the fact that MC/DC is separated from the other components of the generator and therefore its formalization can be handled separately, for instance, by using Z notation, provided that the generator is able to interpret this notation. Another message given by Figure 6 is that the code transformed into a format that can be understood by the generator, must still be consistent with the formal specifications of the program.

There is certainly more work to be done in this subject, however the strengths of the tools and the techniques outlined in this paper will certainly draw a guideline in the production of future tools to cover MC/DC and we continue to explore the use of formal methods to achieve MC/DC-compliant test case generation.

REFERENCES

AC#20-115B (2003). Advisory Circular (AC) # 20-115B, FAA.

Adrion, R., Branstad, M. and Cherniavsky, J., (1982). Validation, Verification and Testing of Computer Software. Computing Surverys, ACM.

Ammann, P., Offutt, J. and Huang, H., (2003). Coverage Criteria for Logical Expression. International Symposium on Software Reliability Engineering (ISSRE '03).

Cavarra, A. , Crichton, C., Davies, J., Hartman, A., Jeron, T. and Monier, L., (2002). Using UML for Automatic Test Generation. Proceedings of ISSTA.

Certification Authorities Software Team (CAST), 2001. Rationale for Accepting Masking MC/DC in Certification Projects.

Chilenski, J.J. and Miller, S.P., (1994). Applicability of Modified Condition/Decision Coverage to Software Testing. Software Engineering Journal.

Chilenski, J.J. and Newcomb, P.H., (1994). Formal Specification Tools for Test Coverage Analysis. The Boeing Company.

Díaz, E., Tuya, J. and Blanco, R., (2004). A Modular Tool for Automated Coverage in Software Testing, Software Technology and Engineering Practice. IEEE CS Press, pp. 234-240.

DO-178B, (1992). DO-178B: Software Considerations in Airborne Systems and Equipment Certification, RTCA, Washington D.C., USA.

Durrieu, G., Laurent, O., Seguin, C. and Wiels, V., (2004), Automatic Test case Generation for Critical Embedded Systems. DASIA'04.

Edvardsson, J., (1999), A Survey on Test Data Generation, ECSEL.

Ferguson R. and Korel B., (1996). *The Chaining Approach for Software test data generation*. ACM Transactions on Software Engineering and Methodology, 5(1):63-86.

Hayhurst, K.,Veerhusen, D., Chilenski, J. and Rierson, L.K., (2001). A Practical Tutorial on Modified Condition/Decision Coverage. NASA.

Jackson, M., (2006). What can we expect from program verification?. Innovative Technology for Computer Professionals 39, no.10, PG 65-71.

Kapoor, K. and Bowen, J.,(2004).Formal Analysis of MCDC and RCDC Test Criteria. London South Bank University.

Korel, B. and Al-Yami, A. M., (1996). Assertion-oriented automated test data generation. Proceedings of the 18th International Conference on Software Engineering, (ICSE), pages 71-80. IEEE.

Prasanna, M., Sivanandam, S.N., Venkatesan, R. and Sundarrajan, R., (2005). A Survey on Automatic Test Case Generation. Academic Open Internet Journal,Volume 15.

Tracey, N., Clark, J., Mader, K. and McDermid, J., (1998). An Automated Framework for Structural Test Data Generation. 13th IEEE International Conference on Automated Software Engineering.