Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Janne Kauttio

MC/DC Based Test Selection for Dynamic Symbolic Execution

Master's Thesis
Espoo, September 19, 2013

Supervisor:	Assoc. Prof. Keijo Heljanko
Advisor:	Assoc. Prof. Keijo Heljanko

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Janne Kauttio		
Title: MC/DC Based Test Selection for Dynamic Symbolic Execution			
Date:	September 19, 2013	**Pages:**	viii + 60
Major:	Theoretical Computer Science	**Code:**	T-79
Supervisor:	Assoc. Prof. Keijo Heljanko		
Advisor:	Assoc. Prof. Keijo Heljanko		

Ensuring the correct operation of a software system is a relevant part of any software development process, but especially important for safety critical software systems used in aircrafts where failures can have fatal consequences and extreme reliability is required. As there is no official record of a software error being the main cause of an aircraft crash up to date, these systems also seem to be very reliable in practice. This success can at least partially be attributed to the aviation software certification process, which places a set of strict requirements on the system that must be taken into account during its development.

One such requirement, applicable only to the most critical systems, is showing complete modified condition/decision coverage on the implementation of the system using tests generated from the system specification. Modified condition/decision coverage is a very demanding form of code coverage, whose purpose is to show both that sufficient testing has been performed to ensure correct operation, and that the implementation is correct in terms of the specification. As generating the tests by hand is very demanding, in terms of this work we study how automation can be used to facilitate this process.

This thesis presents goal constraints, a novel extension to dynamic symbolic execution, which makes automatic generation of a set of tests satisfying modified condition/decision coverage possible. The goal constraints are essentially additional constraints on the values of variables instrumented into the program source code. They can be used during dynamic symbolic execution both to direct the testing process, and to select test cases based on the code coverage they provide.

We present how goal constraints can be automatically generated and instrumented into the source code of a program written in the C programming language, and how support for goal constraints is implemented in an automated software testing tool called LIME Concolic Tester. The implementation is also evaluated, and the advantages and disadvantages of automated test generation in the context of aviation software development are discussed.

Keywords: MC/DC, DSE, code coverage, automated software testing, source code instrumentation

Language: English

Aalto-yliopisto
Perustieteiden korkeakoulu
Tietotekniikan koulutusohjelma

DIPLOMITYÖN
TIIVISTELMÄ

Tekijä:	Janne Kauttio		
Työn nimi: MC/DC-pohjainen testivalinta dynaamiselle symboliselle suoritukselle			
Päiväys:	19. syyskuuta 2013	**Sivumäärä:**	viii + 60
Pääaine:	Tietojenkäsittelyteoria	**Koodi:**	T-79
Valvoja:	Professori Keijo Heljanko		
Ohjaaja:	Professori Keijo Heljanko		

Ohjelmistojärjestelmän oikean toiminnallisuuden varmistaminen on olennainen osa mitä tahansa ohjelmistokehitysprosessia, mutta erityisen tärkeää lentokoneissa käytettäville turvallisuuskriittisille järjestelmille, joissa ongelmilla voi olla vakavat seuraukset. Koska yksikään lentokone ei ole vielä toistaiseksi pudonnut virallisten lähteiden mukaan suoraan ohjelmistovirheen seurauksena, nämä järjestelmät vaikuttavat myös täyttävän niille asetetut luotettavuusvaatimukset erittäin hyvin. Hyvästä menestyksestä voidaan ainakin osittain kiittää lentokoneohjelmistojen kelpoistamisprosessia, joka määrittelee järjestelmille joukon tiukkoja vaatimuksia, jotka on otettava huomioon niiden kehityksen aikana.

Yksi näistä vaatimuksista on täydellinen MC/DC-peittävyys järjestelmän toteutukselle käyttäen testejä, jotka on tuotettu järjestelmän määritelmästä. Tämä vaatimus koskee ainoastaan kaikista kriittisimpiä järjestelmiä, ja sen tarkoitus on osoittaa paitsi että toteutus on määritelmän mukainen, myös että toteutus ei sisällä ei-toivottua toiminnallisuutta. Koska MC/DC-testien tuottaminen käsin on hyvin työlästä, tutkimme tämän työn puitteissa miten automaatiota voidaan soveltaa helpottamaan tätä prosessia.

Tämä työ esittelee tavoiterajoitteet, uuden laajennksen dynaamiselle symboliselle suoritukselle, joka mahdollistaa automaattisen MC/DC-testien tuottamisen. Tavoiterajoitteet ovat olennaisesti ohjelman lähdekoodiin lisättyjä lisävaatimuksia siinä esiintyvien muuttujien arvoille, ja niitä voidaan käyttää dynaamisen symbolisen suorituksen aikana sekä ohjaamaan testausta, että valitsemaan kiinnostavia testejä.

Esitämme miten tavoiterajoitteet voidaan automaattisesti tuottaa ja instrumentoida C-kielisen ohjelman lähdekoodiin, ja miten tuki tavoiterajoitteille on toteutettu automaattiseen ohjelmistotestaustyökaluun nimeltä LIME Concolic Tester. Esitämme myös miten arvion toteutuksesta, ja keskustelemme mitä hyviä ja huonoja puolia liittyy automaattiseen ohjelmistotestaukseen lentokoneohjelmistojen kelpoistamisprosessin näkökulmasta.

Asiasanat:	MC/DC, DSE, peittävyys, automaattinen ohjelmistotestaus, lähdekoodin instrumentointi
Kieli:	Englanti

Abbreviations and Acronyms

API	Application Programming Interface
AST	Abstract Syntax Tree; an abstract source code representation
CPU	Central Processing Unit
CUTE	Concolic Unit-Testing Engine; an automated software testing tool
DART	Directed Automated Random Testing; an automated software testing tool
DO-178B	Software Considerations in Airborne Systems and Equipment Certification; the official documentation of development practices necessary to obtain certification for an aviation software system
DSE	Dynamic Symbolic Execution; an automated directed software testing method
FAA	Federal Aviation Administration
LCT	LIME Concolic Tester; an automated software testing tool
LLVM	Not an acronym (originally Low Level Virtual Machine); a compilation framework
MC/DC	Modified Condition/Decision Coverage; a source code coverage criterion
NASA	National Aeronautics and Space Administration
NIST	National Institute of Standards and Technology
RAM	Random Access Memory
SAGE	Scalable, Automated, Guided Execution; an automated software testing tool

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As various software systems play an important role in virtually every aspect of the modern society, software errors have become something most of us have to deal with on a daily basis. Whereas a bug in the operating system of a mobile phone might lock up the device and force a reboot — resulting in mild annoyance to the user — a failure in a *safety critical* [53] software system controlling the cooling in a nuclear power plant for instance can have more serious consequences. These consequences can be anything from significant financial losses at best to even the endangerment of human lives at worst, as was the case with a series of accidents caused by the Therac-25 radiation therapy machine between 1985 and 1987 [38]. The combination of severe oversights with some of the safety mechanisms in the machine and extremely bad software development practices lead to six cases of massive radiation overdose on patients, which resulted in serious injuries and several deaths. Some less dramatic examples from more recent history include several instances of space exploration equipment being lost as a direct consequence of a software failure, such as the case of Ariane-5 flight 501, which self-destructed only moments after launch in 1996 after an integer overflow veered it off course [18]. Another very memorable incident is NASA's Spirit rover becoming unresponsive shortly after landing on Mars in 2004 [46], even though this time the issue, a problem with the device's flash memory, could be fixed remotely and more severe losses avoided.

According to a study made by the National Institute of Standards and Technology (NIST) in 2002, software errors cost the United States economy 59.5 billion dollars every year [55]. As errors in software systems are very undesirable for aforementioned reasons, significant research effort has been spent on developing tools and practices that help with finding and avoiding them. It is generally recommended that at least some of these quality control activities are included in most software development processes [44]. The most commonly used method for finding and fixing software errors is *software testing* [41]. In short, software testing is the process of obtaining some level of assurance that a piece of software is working correctly by executing it repeatedly in different configurations and making sure it behaves as expected. In order to make software testing more systematic, a concept called *code coverage* can be used to measure the quality of the performed tests by observing how much of the program behaviour is covered by the test executions. The idea was first published in [40], in which the authors demonstrated a method of generating tests for a program based on a tree representation of the program logic. These days code coverage has become more established, and usually refers to the usage of a well defined *coverage criterion*. A more in-depth introduction to software testing with examples of some of the most commonly used coverage criteria is provided in Section 2.1.

Proper software testing is naturally important for all safety critical software systems, but one field where it is of extreme importance is aviation software development. As software failures on aircrafts are extremely undesirable for obvious reasons, all systems on an aircraft must go through a very strict certification process. This process places a set of requirements on the system that must be taken into account during its development, and must be fulfilled in order to eventually obtain certification for the system. The requirements depend on the role of the system on the aircraft, and are listed in a document called *DO-178B* [47]. One particularly interesting requirement, applicable only to the most critical systems, is that a set of tests derived from the specification of the system must achieve complete *modified condition/decision coverage* [15, 25] (MC/DC) on the system implementation. MC/DC is a very demanding form of code coverage, and the purpose of this requirement is to ensure the correct operation of the system by making sure an extremely close match exists between the specification and the implementation. It has been shown that MC/DC is effective at finding otherwise hard to locate errors in satellite software that is very similar to aviation software in terms of safety concerns and requirements [19]. The details of the coverage criterion and its usage in the certification process are covered in Sections 2.2 and 2.3.

There are two significant problems with software testing that should be taken into account when the reliability of a software system is a concern. The first one is that software testing can only be used to find errors; it can not be used to prove that no errors exist. This is an inherent limitation in the way software testing works, and there is little that can be done about it. The reason for this is simply that regardless of the number of correct executions that are observed, incorrect executions might still exists (unless all executions are evaluated, which is an impossible feat in most cases). This fact also implies that in order to obtain any reliable assurance of correctness, the test executions should be chosen carefully to cover as much of the program behaviour as possible, which is something code coverage can help with. If even extensive software testing is not sufficient, further assurance of program correctness can be obtained via *formal methods* such as *model checking* [16]. The idea in model checking is to create a *model* based on the structure of the system and use a specialised tool called a *model checker* to check if certain properties hold in the model. If a property, such as not reaching an erroneous state, does not hold, a counterexample is obtained and can be traced back to an execution in the original system. The downside of model checking is that it is very demanding to use in practice in terms of both computational and professional resources that are required.

The other problem with software testing is its high cost. Software testing is very difficult to do properly, and it takes a significant amount of time during the software development process (some estimates claim up to 50% of total development time [41]). Automation of the testing process can be used to alleviate this issue, and has thus been a topic of intensive research. There are several different ways to automate software testing, one of which is *dynamic symbolic execution* [22, 51] (DSE). In DSE, a symbolic representation of the program is constructed during the testing process, and used to select interesting *input values* that direct the test executions to cover additional program behaviour. The details of the DSE process are explained in greater detail with examples in Section 3.2, and a brief overview on other automated software testing methods and tools is provided in Section 3.1.

Our goal in this work is to consider how automated software testing can be applied to aviation software development, where the requirements on the testing process are very demanding. More specifically, we examine how DSE can be augmented to automatically generate a set of tests satisfying MC/DC for a program written in the *C* programming language [34], through the use of additional constraints in the program source code. Our tool implementation is based on an automated software testing tool called *LIME Concolic Tester* [29, 32] (LCT), which is introduced in greater detail later in Section 3.3. The idea of directing the DSE process to provide additional value is not exactly new, but so far the efforts have been mostly focused on better scalability [10] or faster error detection [60]. Automatic test generation based on code coverage has also been explored before with an method based on model checking [45]. However, because of the nature and the complexity of the coverage criterion, this approach proved to be somewhat impractical for MC/DC in evaluation [26]. As far as we are aware, our method of implementing a test selection mechanism on top of DSE based on additional constraints in the source code with the intent of satisfying a specific coverage criterion is novel.

The rest of this work is structured as follows: Chapter 2 provides an introduction to code coverage on a general level, and presents MC/DC along with a brief coverage of its role in the aviation software certification process. After this, Chapter 3 first introduces a few popular automated software testing tools, and then describes the DSE process through the use of an example. Once the background has been established, Chapter 4 covers how our coverage-based test generation process is designed and implemented, after which a brief evaluation of the implementation is presented in Chapter 5. And finally, a summary of the work along with some thoughts on the philosophical aspects of automated test generation for aviation software and possible directions for future work are provided in Chapter 6.

Chapter 2

Modified Condition/Decision Coverage

The purpose of this chapter is to serve as a general introduction to software testing and code coverage, to define and introduce MC/DC, and to argue why it is an interesting coverage criterion in terms of automated test generation. The chapter is opened with Section 2.1, which introduces several commonly used coverage criteria and provides definitions for concepts used throughout the rest of the chapter. After the background has been established, the formal definition of MC/DC is provided and explained in Section 2.2. The chapter is closed with Section 2.3, which covers how MC/DC is used in practice by taking a closer look at the aviation software certification process. The section also briefly considers some of the implications the role of MC/DC in certification process has on automated test generation.

2.1 Background

Formally, software testing is a process in which the executions of a program are sampled and compared with the specification to find mismatches that can be reported as errors [43]. In practice, this can be translated to running the program with different input values and making sure it does what it is supposed to. Even though software testing can never be used to guarantee the absence of errors in a program, short of executing the program with all possible combinations of input values and verifying the correct operation in each case, it is a practical and widely used tool for improving the quality of software systems. As software testing in itself is a huge area of research inside the field of computer science, providing extensive coverage of all its aspects is not the intent of this introduction. However, it should be made clear that in terms of this work we are specifically interested in automated *unit testing* of *white-box* software systems. White-box testing, in contrast to *black-box* testing, happens in a setting where complete information on the implementation details of the system (i.e. the source code) is available. This is different from black-box testing, in which the system is treated as a "black-box" and analysis based on the structure of the system is considerably more difficult. The definition of unit testing varies slightly depending on the source, but it usually refers to the process of testing an individual software component as completely as possible with the intent of finding errors in the program logic. [41]

As covering all executions of a program during the software testing process is often difficult, code coverage can be used to measure how well the program source code has been tested. The testing process (a set of program executions) is said to achieve certain code coverage, if the executions in the test set cover a sufficient subset of all possible executions of the program. What constitutes as "sufficient" actually depends on the selected coverage criterion, and it can be practically anything from covering all individual statements to covering all execution paths in the program. The basic idea behind code coverage is that the test executions can be divided into groups of similar executions based on some property they have in common. If the testing process succeeds in covering at least one execution from each group, it can be considered to cover the whole program in terms of that property. This means that, providing the property is something sensible, a set of tests that satisfies a coverage criterion has a better chance of finding an error in the program than a set of tests that does not. Satisfying more complex coverage criteria requires more work during the testing process, but it also usually improves the probability of finding errors. [43]

Since possibly the best way to clarify any complicated concept is through examples, the rest of this section is dedicated to presenting a selection of commonly used coverage criteria. Each criterion is first defined formally, after which an example test set satisfying the criterion for a running example program, available in Figure 2.1, is provided. This program takes three input values as parameters, and returns true if either the first parameter or both the second and the third parameter are positive. If this is not the case, the program returns false. Each individual test case in the test set is given as a combination of values for the input parameters x, y and z. Even though the program might seem very simple, the underlying logic is sufficiently complex to demonstrate the most important differences between the coverage criteria presented in this section. It should also be noted that since the input parameters are integer-valued, verifying the correct operation via exhaustive testing (verifying all possible executions) is infeasible even for a program as simple as this. Assuming 32-bit integers, there is a total number of $(2^{32})^3$ different combinations of input values, all of which would have to be tested in order to verify the correct outcome in each case. (Strictly speaking, this is not exactly true in white-box testing as a better estimate for the number of required test executions can be obtained through analysis of the program structure. However, the number of execution paths is still extremely high for most practical programs, which will become more apparent later.)

```
bool example(int x, int y, int z) {
    bool result = false;
    if (x > 0 || (y > 0 && z > 0)
        result = true;
    return result;
}
```

Figure 2.1: A simple example program

2.1.1 Statement Coverage

Definition 1. A testing process provides *statement coverage* for a program, if and only if each executable statement in the program is executed at least once during the testing process. [43]

Statement coverage (Definition 1) is the simplest and the least demanding of the commonly used coverage criteria. For the example program (Figure 2.1), complete statement coverage can be achieved with just one test, as seen in Table 2.1. The limited usefulness of statement coverage as an assurance of correctness for a program is already apparent with a program as simple as this, as not even both return values are covered by the test set.

x	y	z	Return value
1	0	0	true

Table 2.1: Statement coverage for example

Despite its limitations, statement coverage also has its uses. For instance, statement coverage can be used during the testing process to discover the presence of *dead code* in a program. Dead code (also known as unreachable code) is a part of the program source code that is never executed, and as such an indication of a potential problem in either the design or the implementation of the program. Even though analysing a program for the presence of dead code is an undecidable problem (see [43]), meaning that it can not be done conclusively, incomplete statement coverage can be used as an indication of it. When a part of the program is not executed during the testing process, it means that either the part is unreachable, or that the testing process is not sufficiently thorough to cover that part of the program. In either case, a potential problem in either the program or the testing process is uncovered.

2.1.2 Branch Coverage

Definition 2. A testing process provides *branch coverage* for a program, if and only if each *branch* in the program is covered at least once during the testing process. [43]

In contrast to statement coverage, branch coverage is already far more efficient in covering the different behaviours of a program. A branch in Definition 2 is a point in execution where the next statement or the set of statements to be executed is chosen from two or more alternatives, such as a if-then-else or a switch-case statement. For the example program (Figure 2.1), this means that in the test set there must be at least one test run where the then-branch of the if-statement is taken, and at least one where it is not. This can be done for instance by varying the value of the parameter x in a way which causes the boolean expression to evaluate to both true and false, as seen in Table 2.2.

x	y	z	Return value
0	0	0	false
1	0	0	true

Table 2.2: Branch coverage for example

Even though branch coverage does better than statement coverage, the effect of parameters y and z on the return value in the example program are not necessarily covered by the test set. This is because the boolean expression is structured in a way which makes it possible for the parameter x to make it true regardless of the values of the other parameters. Thus some of the intended program behaviour is still left uncovered by the test set.

2.1.3 Path Coverage

Definition 3. A testing process provides *path coverage* for a program, if and only if each *execution path* in the program is explored during the testing process. [43]

Path coverage is considerably more demanding than statement and branch coverage. An execution path in Definition 3 refers to the sequence of statements and branching points encountered, as well as the branching decisions made during a single execution of the program. In other words, this means a path from an *entry point* to an *exit point*, with each branching decision considered to be a part of the path.

As is seen in Table 2.3, for the example program (Figure 2.1) path coverage is actually provided by the same set of tests that provides branch coverage. This follows from the fact that there is only one branch and thus only two different execution paths in the program. The difference between these two code coverage criteria becomes more apparent when looking at a slightly more complex variant of the running example, available in Figure 2.2. The new program does essentially the same thing as the old program; the only difference between them is the structure of the boolean expression. Now branch coverage can again be achieved with only two test cases (Table 2.5), but in order to achieve complete path coverage, four test cases are necessary. This is because all four different ways of evaluating the boolean formulas must be covered (Table 2.6). For reference, statement coverage on this program can again be achieved with only one test case (Table 2.4).

x	y	z	Return value
0	0	0	false
1	0	0	true

Table 2.3: Path coverage for example

Even though path coverage is effective in covering program behaviour, its biggest limitation is that the number of different execution paths for a given program is often enormous, possibly infinite if certain kinds of loops (e.g. while statements that never terminate) are involved. In addition, not all execution paths in the program are necessarily even executable in practice. This means that achieving complete path coverage, even when technically possible, is difficult as the number of required test cases is often very large. Path coverage is of special importance in this work, as it is the code coverage provided by the dynamic symbolic execution process (see Section 3.2).

```
bool branchexample(int x, int y, int z) {
    bool result = false;
    if (x > 0)
        result = true;
    if (y > 0 && z > 0)
        result = true;
    return result;
}
```

Figure 2.2: A simple example program with multiple branches

x	y	z	First branch	Second branch	Return value
1	1	1	true	true	true

Table 2.4: Statement coverage for branchexample

x	y	z	First branch	Second branch	Return value
0	0	0	false	false	false
1	1	1	true	true	true

Table 2.5: Branch coverage for branchexample

x	y	z	First branch	Second branch	Return value
0	0	0	false	false	false
0	1	1	false	true	true
1	0	0	true	false	true
1	1	1	true	true	true

Table 2.6: Path coverage for branchexample

2.1.4 Condition/Decision Coverage

Condition/decision coverage is a complicated and perhaps slightly artificial coverage criterion, which is covered here mostly because it is the basis for MC/DC, presented later in Section 2.2. Condition/decision coverage is less straightforward than the aforementioned coverage criteria, and in order to define it properly, we must first define what conditions (Definition 4) and decisions (Definition 5) mean.

Definition 4. *Condition* is a boolean-valued expression which does not contain any boolean operators. This means that a condition can not be broken down into simpler boolean-valued expressions. [13, 15, 25]

Definition 5. *Decision* is a boolean-valued expression with zero or more boolean operators. A decision with zero boolean operators is a condition. If a condition appears more than once in a decision, each occurrence is a distinct condition. [13, 15, 25]

According to above definitions, `x > 0`, `y > 0` and `z > 0` in the simple example program (Figure 2.1) are conditions, and the whole expression `x > 0 || (y > 0 && z > 0)` is a decision. In order to cover every condition and every decision, the testing process must show that each condition and decision evaluates to all possible values. Formally, condition/decision coverage can be defined as follows (Definition 6):

Definition 6. A testing process provides *condition/decision coverage* for a program, if and only if all entry and exit points in the program are covered, and every condition and decision in the program evaluates to both true and false at least once during the testing process. [25]

For the example program (Figure 2.1), a test set satisfying condition/decision coverage is given in Table 2.7. All individual condition evaluations for each test are included in the table. This example illustrates the limitations of condition/decision coverage rather well. Even though the test set technically covers all branches (decision coverage) in the program and even shows that all atomic boolean expressions evaluate to both values (condition coverage), it still fails to show the effect each individual condition has on the program logic. Imagine a scenario where the first disjunction (`||`) in the example program would be accidentally replaced with an conjunction (`&&`), resulting in the expression `x > 0 && (y > 0 && z > 0)`. If the testing process relies solely on condition/decision coverage, the same test would still provide complete code coverage and this error would go completely unnoticed, even though the behaviour of the program would be completely different.

x	y	z	$x > 0$	$y > 0$	$z > 0$	Return value
0	0	0	false	false	false	false
1	1	1	true	true	true	true

Table 2.7: Condition/decision coverage for example

Another coverage criterion, *multiple condition coverage*, requiring the testing process to cover all possible combinations of values of conditions in each decision, could be used to solve the problem experienced with condition/decision coverage. However, there is a bigger problem associated with multiple condition coverage, which is the extremely large number of tests it requires. If a program contains a decision with n conditions, 2^n distinct tests is necessary to show complete multiple condition coverage for that condition alone. This fact makes this particular coverage criterion impractical in most applications. [25]

2.2 Definition

Modified condition/decision coverage is a complex code coverage criterion, which can be used to provide some of the benefits of multiple condition coverage without as large blowup in the size of the test set. As the name suggests, MC/DC is basically condition/decision coverage slightly modified with an additional *independence requirement.* This requirement makes sure that in addition to everything required by condition/decision coverage, the testing process must also show that each condition in each decision actually affects the outcome of the decision. Formally, MC/DC can be defined as follows (Definition 7 according to [15, 25]):

Definition 7. In order to provide *modified condition/decision coverage*, the testing process must fulfill each of the following requirements:

(i) Every entry and exit point in the program must be covered at least once.

(ii) Every condition in the program has taken all possible outcomes at least once.

(iii) Every decision in the program has taken all possible outcomes at least once.

(iv) Every condition in a decision has been shown to independently affect the outcome of the decision.

The discussion here focuses mainly on the fourth requirement, as the first three requirements follow directly from condition/decision coverage and have already been covered in Section 2.1.4. The key idea behind the independence requirement is that each condition in a decision should be there for a reason. This means that for each condition in a decision, all other conditions in the decision should have at least one combination of values such that changing only the value of the selected condition changes the value of the whole decision. Two value combinations that show the independent effect of a condition in a decision (by changing only the value of the condition and showing it changes the value of the decision) are called an *independence pair* [25] for that condition. In order to show the individual effect of each condition in a program, the test set must include independence pairs for all conditions. This means that the number of tests required in order to provide MC/DC on a program is linear in the number of conditions, which is significantly less than the exponential number of tests required my multiple condition coverage. [15]

Table 2.8 shows a test set satisfying MC/DC on the example program (Figure 2.1). As the program only has one entry and exit point, and as each condition and decision is true and false at least once during the testing process, MC/DC Requirements (i), (ii) and (iii) are clearly satisfied. In order to satisfy Requirement (iv), the test set should contain an independence pair for each of the three conditions, labelled with X for `x > 0`, Y for `y > 0` and Z for `z > 0`. The pairs can be formed as depicted in the last column (notice here that the pairs may actually overlap as is the case with the third test), so the independent effect of each condition is shown and MC/DC is provided.

x	y	z	x > 0	y > 0	z > 0	Decision	Pair
0	0	0	false	false	false	false	X
1	0	0	true	false	false	true	X
0	1	1	false	true	true	true	Y,Z
0	0	1	false	false	true	false	Z
0	1	0	false	true	false	false	Y

Table 2.8: Modified condition/decision coverage for example

There is a common misconception regarding the meaning of a decision in the context of MC/DC, which should be clarified here. The confusion stems from the fact that the term “decision” is traditionally associated with a branching point, whereas in this context it actually refers to *any* boolean-valued expression in the program. Unless the correct interpretation is used, the amount of program logic covered by MC/DC can be severely limited in some cases. This issue can be illustrated rather nicely with yet another variant of the running example, available in Figure 2.3. In this program, the boolean expression in the branching point is separated into two sub-expressions (decisions), A and B. Since the program logic is actually the same as in the original program (Figure 2.1), the test set providing MC/DC for this program should be very similar to the one presented in Table 2.8. If we use the correct interpretation of decision this is indeed the case, as all of the following requirements must be satisfied at least once during the testing process:

1. Condition `y > 0` has the correct independence effect on decision *A*.
2. Condition `z > 0` has the correct independence effect on decision *A*.
3. Decision *A* has been assigned both true and false.
4. Condition `x > 0` has the correct independence effect on decision *B*.
5. Condition `A` has the correct independence effect on Decision *B*. Note here that even though *A* in the program is also a decision, `A` in this context is a condition because it is an atomic boolean-valued sub-expression of decision *B*.
6. Decision *B* has been assigned both true and false.

The different decision structure changes the actual value combinations slightly from Table 2.8, but the individual effect of each condition is still shown correctly. Now if we would use the incorrect interpretation of a decision, limiting our consideration to just the decision in the branching point, only the last requirement (6) would have to be satisfied during the testing process in order to provide "complete" MC/DC. In this case MC/DC reduces to essentially branch coverage, and the individual effect of each condition in the program is not shown. Thus, It is important that all boolean expressions in the program are treated as decisions in order to make sure MC/DC works as intended for all kinds of programs. More detailed coverage of this issue is available in [13].

```
bool decisionexample(int x, int y, int z) {
    bool result = false;
    bool A = y > 0 && z > 0;
    bool B = x > 0 || A;
    if (B)
        result = true;
    return result;
}
```

Figure 2.3: A simple example program with multiple decisions

2.2.1 Different Variants

There is one detail in the literal definition of MC/DC regarding the meaning of a condition that causes some problems when the code coverage criterion is used in practice. Looking back at Definitions 4 and 5, we can see that each atomic boolean valued sub-expression in a decision is considered to be a *distinct* condition, regardless of the actual structure of the expression. This means that if the underlying sub-expressions happen to be similar or refer to the same variables, we obtain two "different" conditions whose values can not be changed independently. If changing the value of one condition affects the value of another condition, we say that those two conditions are *coupled.* [25]

If a decision in a program has coupled conditions, satisfying the independence requirement according to the literal definition of MC/DC becomes difficult. This is because the individual effect of each condition must be shown with an independence pair during the testing process, and constructing an independence pair where only the value of one condition changes is not possible if the condition is coupled. This form of MC/DC is called *unique-cause MC/DC* [14], and providing it for programs with coupled conditions is actually impossible.

Unique-cause MC/DC has also other issues in addition to the limited support for boolean expressions, again caused by the strict requirements on the independence pairs. If we take another look at the decision in the example program in Figure 2.1, we can observe that the individual values of the conditions in the sub-expression `(y > 0 && z > 0)` do not actually affect the outcome of the decision unless they are both true. Using this observation, the first test in Table 2.8 could actually be removed, as the independence effect of the condition `x > 0` can be demonstrated using the second test and either of the last two tests instead. However, since the definition of an independence pair strictly requires that only the value of one condition changes, the first test is necessary for unique-cause MC/DC.

Another form of MC/DC, called *masking MC/DC* [14], has been developed to work around the aforementioned issues. The idea behind masking MC/DC is that the independent effect of a condition can be shown with an independence pair where the values of other conditions are also allowed to vary, if they do not have an effect on the truth value of the decision (i.e. they are /masked]). In the previous example, the values of `y > 0` and `z > 0` can vary freely in the independence pair for `x > 0`, as long as the sub-expression `(y > 0 && z > 0)` remains false, since then the outcome of the decision is decided by the value of `x > 0` alone as is intended. The downside of this is that recognising when a condition is masked in a decision requires some analysis on the structure of the decision, which makes finding the independence pairs potentially more difficult.

Coupled conditions are not a problem with masking MC/DC, as they can be treated like all other conditions. An important thing to note is that the definition of a condition does not change, so the independent effect of each coupled condition must still be shown during the testing process. Masking MC/DC only makes it possible to find independence pairs for coupled conditions; it does not guarantee that they exist. Also, because of the relaxed requirements on the independence pairs, there are more potential independence pairs that can be used to satisfy the independence requirement with masking MC/DC than with unique-cause MC/DC. What this means is that we can choose the independence pairs with as much overlap as possible for the conditions, resulting in a smaller test set. This effect was already visible to a degree in the previous example, where we could remove one test if masking MC/DC was used.

The downside of masking MC/DC is that the probability of finding an error in a decision (e.g. a wrong boolean operator) can be slightly lower than with unique-cause MC/DC. The reason for this is not covered here as the analysis is rather involved and not very relevant in the context of this work, but more details are available in [14]. Whether masking MC/DC is a suitable replacement for unique-cause MC/DC or not is somewhat debatable, but the consensus [12, 14] seems to be that this is indeed the case.

2.3 Usage

All aforementioned code coverage criteria are commonly used in software development, but what makes MC/DC especially interesting to us is its position in the aviation software certification process. In this section we take a closer look at this certification process to see how MC/DC is used in practice, and to find out what consequences this has on automated coverage-driven test generation.

Covering all the details of the aviation software certification process is not possible in the scope of this work, as the process is quite complex and the details may vary from aircraft to aircraft. The basic idea is that the regulations and possible special conditions applicable to an aircraft model or type are initially defined in *certification basis*, which is established by the certification authority (i.e. Federal Aviation Administration or FAA in the United States) together with the aircraft developer when the aircraft model or type is submitted for certification. This means that a software system is always certified as a part of the complete aircraft. The aircraft developer must then provide *means of compliance*, which defines how the certification basis is satisfied by the development process. The aviation software certification process is largely based on standards, and historically very effective in producing safe software systems. In fact, no aircraft crashes can be attributed to software errors up to date. [49]

One important step in the aircraft design process is *safety and hazard analysis*. Each system on the aircraft performs a specific function. This means that a failure in a system will lead to either malfunction, unintended function or loss of function on the aircraft. Safety and hazard analysis considers these failure states in terms of *failure conditions*, ranging from “catastrophic” (i.e. aircraft crash) through “hazardous”, “major” and “minor” to “no effect”, depending on the effect they have on the safety of the aircraft, crew or passengers. The primary goal of the analysis is to help in the system design process, so the number and severity of failure conditions can be minimised. This is done by refining the design of a system according to the results of the analysis, and increasing the reliability of subsystems (by providing backup systems for instance) as necessary. When the design of a component is no longer refined for safety, it is required that the implementation of the component is *correct* in terms of the specification (meaning it does what it is supposed to do). The current guidelines for ensuring the correctness of the implementation are defined in DO-178B [47], “Software Considerations in Airborne Systems and Equipment Certification”, which can be considered to be the “standard” for aviation software development and certification. [49]

DO-178B outlines several software development and analysis processes that can be used by aviation software developers as means to obtain approval for a software system. A new version of the document, *DO-178C* [48], has been recently released in order to bring DO-178B up to date with current software development practices. However, as the changes in the update are not covered extensively in literature as of this writing, and verifying the exact nature of the differences between the old and the new version is difficult (neither version is publicly available), we shall focus on DO-178B instead. DO-178B identifies five different *design assurance levels* (or *software levels*) ranging from A (highest) to E (lowest), and a set of *objectives* that the software development and analysis processes should satisfy for each level. The level of a software system is determined by the failure condition associated with the system, level A corresponding to “catastrophic”, level B to “hazardous” and so on. The specification of a software system can be expressed as a set of *requirements*, describing the features of the system and how it should behave. One basic objective in DO-178B which is used to ensure the correct implementation of a level A-D software system is showing *requirements coverage* for the system, which basically consists of generating a set of tests from the requirements and using it to show that the implementation satisfies the requirements. [25]

The biggest limitation with requirements coverage is that it is not alone sufficient to guarantee that a software system has been tested thoroughly. This is because the set of requirements from which the test set is generated might not be a complete representation of all possible behaviours of the implementation, or the implementation might simply contain unintended functionality. This is where another objective, showing *structural coverage* for the system, comes in. Structural coverage is somewhat similar to requirements coverage, but now it must be shown that the test set generated from the requirements provides certain code coverage on the implementation. The objective is to show that the test set exercises the structure of the program code to a sufficient level, and also to reveal parts of the program code which are not covered by the test set. Together requirements and structural coverage provide a great level of confidence that a software system both does what it is supposed to do and does not contain any unintended functionality. An additional benefit of structural coverage is that it implicitly enforces a relatively close match between the requirements and the implementation, as all tests are traceable both ways between the two. [25]

The type of code coverage criterion used with structural coverage varies with the level of the software system which is being tested, ranging from statement coverage for level C systems to MC/DC on level A systems. Analysing the structural coverage of a test set generated from the requirements by hand is very tedious, but possible by for example using the method based on *boolean circuits* described in [25]. Automating the complete test generation and coverage measurement processes might seem like an enticing idea, but is actually pretty dangerous unless the capabilities and limitations of the used tools are well known in advance. Most notably, all the confidence provided by requirements and structural coverage does not mean much if the code coverage is not measured correctly. This is the primary reason why we, in the scope of this work, are only concerned with automating the test generation part of the process. Our idea is that the generated test set can be used as a sensible starting point for the testing process, but also that we do not have to provide any guarantees on the completeness of the test set, as the coverage measurement part must still be performed separately.

Chapter 3

Dynamic Symbolic Execution

This chapter provides the background for our coverage-based test generation process by introducing the method and tool our solution is based on, namely dynamic symbolic execution and LIME Concolic Tester respectively. As a small introduction to automated software testing and to provide an overview on the related work on the field, Section 3.1 first covers some other well known automated software testing tools and the methods they utilise. After the introduction, DSE is presented through an example in Section 3.2, and the important details of LCT are covered in Section 3.3.

3.1 Automated Software Testing

In order to properly discuss automated software testing, a couple of important definitions are necessary. The first thing that should be clarified is the distinction between "automated software testing" and "test automation". Even though the terms can sometimes be used somewhat interchangeably, the latter usually refers to the automation of the actual testing process, i.e. the execution of a predetermined set of test cases, whereas the former also covers the automatic generation of said test cases. Since we are not primarily interested in the practical aspects of software testing in terms of this work, the discussion in this section is mostly limited to the test generation part of the process. Another important notion is the difference between *static* and *dynamic* analysis of a program. Static analysis is based on the structure of the program and is performed on the program source code (or sometimes object code), and dynamic analysis is based on the program execution. These two practices are far from exclusive however, and as a matter of fact most of the methods presented here are a combination of the two. It should also be noted that the availability of source code is crucial for most automated software testing methods, which is the primary reason why we are mostly dealing with white-box testing (as mentioned in Section 2.1).

The problem of automating the software testing process can be broken down into several parts. First is the extraction of the program interface and identification of the so called *inputs* or *test inputs*, which determine what the execution path during the test execution will look like. Even though any value of a variable that is not decided exclusively by the execution history of the program can technically be considered an input, sources of nondeterminism that are not part of the program interface (such as values from a random number generator) are usually left outside the consideration since they are more complicated to identify and handle. Some automated software testing tools provide more automation in the program interface extraction than others, but in the scope of this work the inputs are always indicated explicitly to avoid confusion. Second part of the process is the selection of interesting input values that exercise as much of the program structure as possible. This is the most interesting part of automated software testing, and also the part that differs the most between different methods. The third and final part of the process is the actual execution of the program with the selected input values, which usually also includes additional analysis of the program structure which is necessary for the selection of input values for subsequent test executions.

Possibly the simplest way of doing automated software testing is randomization of the input values, also known as *random testing* or *fuzzing* [8]. Despite its apparent simplicity, random testing can be effective in finding actual errors [20] and is generally useful thanks to its applicability to all kinds of programs (it also works with black-box testing). However, its biggest limitation is that it struggles with certain kinds of relatively common branching structures, and is thus far from excellent in achieving good code coverage [42]. For instance, an if-statement with a comparison between an integer variable and a specific value, such as `x == 5`, is rarely covered in random testing, as the range of possible values for `x` is huge and randomly selecting any single one is extremely unlikely. A tool called *DART* [22] (*Directed Automated Random Testing*) solves this issue with a clever combination of *concrete* and *symbolic* execution of the program. The approach utilised by DART is commonly known as dynamic symbolic execution or *concolic testing* (a combination of *conc*rete and symb*olic*), and is demonstrated in greater detail through an example in the next section. In short, *symbolic constraints* on the input values are collected during concrete executions based on the encountered operations, after which they are combined into *path constraints*. These path constraints can then be solved with a *constraint solver* to obtain new input values that should direct the execution down a particular execution path. This idea is very similar to the traditional *symbolic execution* [36], the primary difference being the fact that in this case the path of the concrete execution is always followed [51].

In practice DART works by first using static analysis to instrument the target program source code written in the C programming language with additional statements that enable the symbolic execution. After this the program is executed, first with random input values, and then with values obtained from the path constraints collected during previous executions. During the execution, DART keeps track of both the concrete and the symbolic value of each variable. The concrete value is obtained from the actual execution, and the symbolic value is constructed based on the encountered operations. Symbolic variables are managed based on their locations in memory, which allows for similar handling of constants, expressions and pointers. The downside of this method is that it does not allow expressions with side effects or pointer dereference when the value of a pointer depends on an input. If the symbolic evaluation of a variable fails for some reason, the concrete value of the variable is used instead. The testing process terminates when an error is found or all execution paths have been covered, and is restarted with new random input values if a situation that might lead to incompleteness is encountered. [22]

The basic ideas from DART are also utilised in several other automated software tools with their own variations and additions. *CUTE* [52] (*Concolic Unit-Testing Engine*) is one such tool. Even though CUTE is similar to DART in many respects, namely it also works with C programs and is based on dynamic symbolic execution with source code instrumentation, it supports a wider variety of programs thanks to several key improvements. Instead of plain memory addresses, the inputs in CUTE are *logical input maps*, which can be used to represent any finite memory graph as a collection of symbolic variables, and thus allow the symbolic execution to tackle for instance dynamic data structures with complicated pointer operations. Another important improvement in CUTE is the separation of pointer and integer constraints, which allows for more simplified constraints and therefore makes the symbolic execution more efficient. A slightly more recent take on automated software testing is provided by *SAGE* [23] (*Scalable, Automated, Guided Execution*) and *Pex* [57], both of which improve the dynamic symbolic execution process with additional heuristics. SAGE is heavily focused on performance; it works on the level of *x86* instructions [27], uses a search heuristic to quickly maximise code coverage, and includes several optimisations that make it able to handle large applications. Pex also improves code coverage with heuristics, but the main thing that makes it different from the other tools is its close relationship with the constraint solver *Z3* [17], which allow for efficient reasoning of *safe* and *unsafe* features in *.NET* [56] programs.

The traditional symbolic execution [36], on which dynamic symbolic execution is also based on, is another popular approach to automatic generation of tests for software systems. As mentioned previously the primary difference between dynamic and non-dynamic symbolic execution is the utilisation of concrete executions to explore the program structure. Although this difference might seem relatively minor, it actually has significant effect on how these methods actually work in practice. More specifically, with plain symbolic execution the program must be executed symbolically from the start, which requires more careful management of symbolic values as the execution does not have concrete values to fall back to if something goes wrong. At the start of the symbolic execution the symbolic value of each input is unrestricted, and as the execution proceeds the value is refined with additional constraints based on the encountered operations. If the program branches, the symbolic execution must also fork and consider both cases separately. Notice that this behaviour is different from dynamic symbolic execution, where the execution always follows the path of the concrete execution. Whenever a bad state is observed in the program, the constraints collected on that execution path can be solved to obtain concrete input values that lead to said bad state. Conversely, if the constraints are not satisfiable, the error is not reachable with any input values (providing the symbolic execution is correct). There are a few tools based on this idea, the most popular of which probably being *KLEE* [11] and *Java PathFinder* [58]. Both of these tools work on the *object code*, or compiled source code, representation of the program, namely on *LLVM* [4, 37] *bitcode* and *Java* [24] *bytecode* respectively. The defining features of the tools include clever constraint and environment management methods to heavily increase performance in KLEE, and the use of model checking to provide support for advanced programming language features in Java PathFinder.

3.2 DSE Through an Example

As mentioned previously, dynamic symbolic execution is an automated software testing method whose purpose is to discover errors in a program by automatically exploring as many of execution paths as possible. In practice this is done by executing the program simultaneously concretely and symbolically, first with random and then with carefully selected input values, and collecting information on the structure of the program during the execution. This information, represented as symbolic constraints, can then be used to select new input values which lead the execution down a particular path. [51]

As this process is rather involved, it is best illustrated with an example.

Figure 3.1 represents the source code and the control flow structure of a simple example program with two error conditions and one input value. As was already discussed, the input values and the error conditions are indicated explicitly in the example for the purpose of clarity. This does not limit our consideration of the testing process in any meaningful way, since the interface extraction process required to identify them automatically is not strictly a part of DSE. As a matter of fact, manual definition of at least the input values is actually required by most automated software testing tools. In this example the input value is obtained from the `input()` method, and the error conditions, which can be considered similar to assertion failures, are indicated with the `error()` method. Note that the first error in the program is actually unreachable regardless of the value of the input, and that the second error is only encountered with a very specific value. The purpose of this is to demonstrate how these different situations are handled by DSE, and to show how the method is able to rather easily find errors that are nigh impossible for e.g. random testing to detect. In the example, the symbolic constraints are expressed in terms of traditional logic and arithmetic operations. A more detailed overview of the constraint language is available for instance in [29].

The progress of the testing process with DSE is depicted in Figure 3.2. Since nothing is known before the first execution, the initial input value is chosen at random and the initial path constraint is empty. Assume that in this case the input value happens to be -3, which means that the first execution path is 2, 3, 4, 7, 8, 14 and the return value is 9 (Figure 3.2a). The symbolic constraints collected during the execution are listed next to the corresponding edges in the graph. Now the testing process observes two uncovered branches, 5 and 9, and attempts to cover them systematically. Assume that the path to 5 is attempted first. The path constraint for the next execution is formed based on the symbolic constraints on the path 2, 3, 4, 5, which results in $(x_0 = \mathit{input}_0) \wedge (y_0 = x_0 * x_0) \wedge (y_0 < 0)$. This constraint is unsatisfiable, which means that the path is unreachable regardless of the value of the input and does not have to be considered again (Figure 3.2b). The path to 8 however, with the corresponding path constraint $(x_0 = \mathit{input}_0) \wedge (y_0 = x_0 * x_0) \wedge \neg(y_0 > 0) \wedge (x_1 = x_0 + 2) \wedge (x_1 > 0)$, is satisfiable with any input value ≥ -1. Assume that the value 1 is chosen, so the next execution path is 2, 3, 4, 7, 8, 9, 12 and the return value is 3 (Figure 3.2c). Since a new uncovered branch, the path to 9, is found, the path constraint for the next execution is $(x_0 = \mathit{input}_0) \wedge (y_0 = x_0 * x_0) \wedge \neg(y_0 > 0) \wedge (x_1 = x_0 + 2) \wedge (x_1 > 0) \wedge (y_0 = x_1)$. This constraint is satisfiable with the input value 2, which results in the execution path 2, 3, 4, 7, 8, 9, 10 (Figure 3.2d). This execution finds and error in the program, and the testing process is terminated.

```
int example() {
    int x = input();
    int y = x * x;
    if (y < 0) {
        error();
    }
    x = x + 2;
    if (x > 0) {
        if (x == y) {
            error();
        }
        return x;
    }
    return y;
}
```

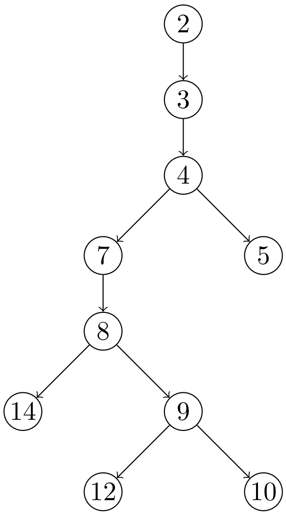
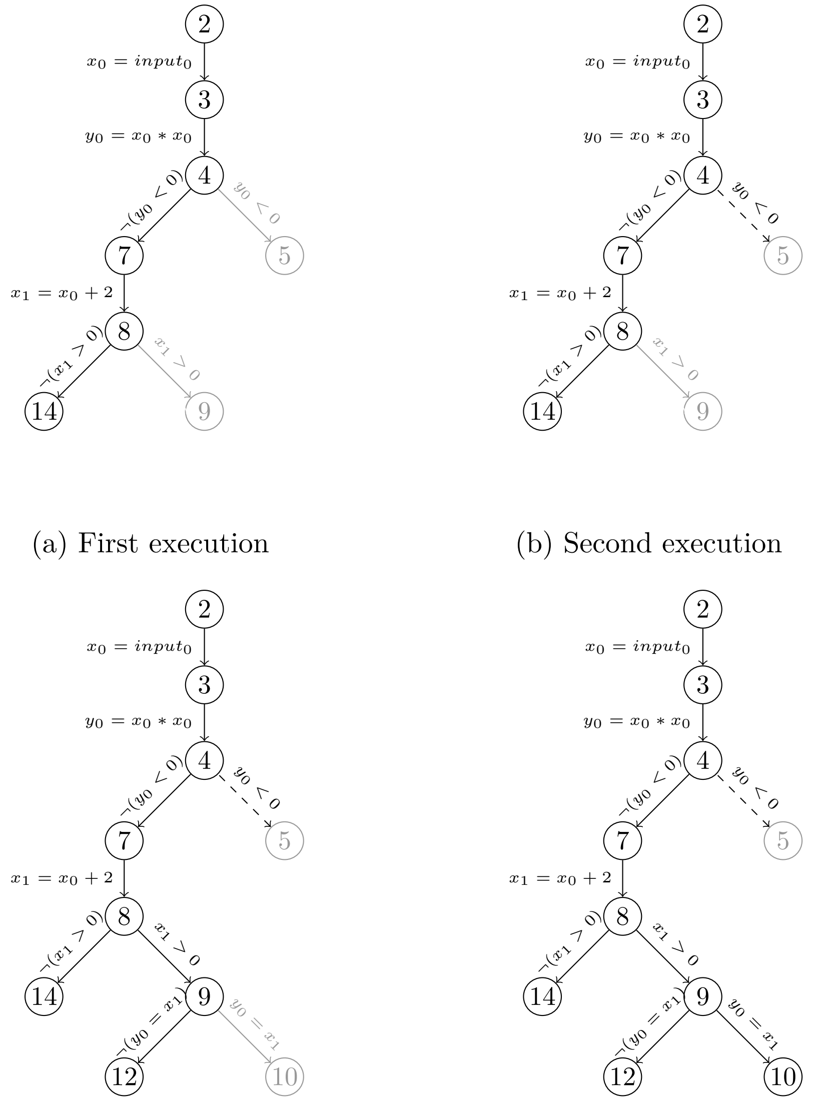


Figure 3.1: An example program with graph representation

The biggest limitation of DSE is that if the symbolic execution is not able to track the concrete execution for some reason, then the testing process can not guarantee complete coverage of all the execution paths in the program. This situation can arise if the program does an operation that is not supported by the symbolic execution, or executes a library function that has not been instrumented. The traditional solution in this case is to simply fall back to the concrete input values and proceed with the concrete execution. Fortunately these situations can usually be recognised during the testing process, and the user can at least be made aware of the incomplete coverage. Another related problem is that a branch that is unreachable during the testing process (i.e. the corresponding path constraint is unsatisfiable) might not necessarily be unreachable in practice if the program has sources of nondeterminism that are not treated as inputs. This only happens if the inputs are not defined correctly and is thus usually an user error, but it is still very possible in practice as sometimes some sources of nondeterminism are hard to recognise if the program uses values from a random number generator for instance. However, if the whole program is within the scope of the symbolic execution and all inputs are defined correctly, DSE should be able to find any error that is reachable with any combination of input values (and thus provide exhaustive testing for the program).

Another common issue with DSE is again the fact that the number of different execution paths in a program can be extremely large. This means that even when DSE is technically not limited by the aforementioned issues, complete coverage might still be computationally infeasible for any reasonably



(a) First execution

(b) Second execution

(c) Third execution

(d) Fourth execution

Figure 3.2: Progress of the DSE process

complicated program. This problem is amplified by the fact that DSE is unable to recognise a situation where two distinct execution paths converge to the same program state, and instead treats all execution paths individually. These problems can be circumvented to a degree with heuristics methods that select execution paths that increase code coverage quickly during the testing process (as is the case with some of the testing tools presented in Section 3.1), but the fact remains that complete coverage is still difficult to achieve.

3.3 LIME Concolic Tester

LIME Concolic Tester is yet another automated software testing tool that is based on DSE. LCT is of special importance in the context of this work, as it is the tool our coverage-based test generation process is based on. This section introduces the primary features of the tool, and provides the background on the relevant implementation details that are necessary to understand how the test generation process presented in the next chapter works in practice.

At its heart, LCT is very similar to the automated software testing tools presented earlier. It is developed in the Department of Information and Computer Science in the Aalto University School of Science, and its primary features include automatic instrumentation of the program that enables symbolic execution, a distributed architecture that supports several concurrent test executions, and integration with the bitvector constraint solver *Boolector* [9] that makes more precise symbolic execution possible [29, 32]. LCT can technically work with both Java and C programs, but several of its advanced features currently only work with Java, as the support for C is a very recent addition and thus partially incomplete. LCT also supports automated testing of multithreaded Java programs with a modified *dynamic partial order reduction* algorithm [50], which has also recently been extended with support for *unfoldings* [33]. The tool has been evaluated as a part of a study that compares the DSE approach with random testing on a set of smart card Java applets, in which DSE was found to be significantly more effective [30]. LCT is open source and freely available as a part of the *LIME TestBench* [3] toolkit under the *MIT* license. In addition to LCT, the toolkit also contains a runtime monitoring tool for Java and C programs that is based on the *LIME interface specification language* [31]. As our goal in this work is to provide an automated test generation process specifically for C programs, the implementation details provided in the next section also focus on the C programming language support in LCT.

3.3.1 Implementation

Perhaps the most defining feature of LCT is its distributed architecture, which allows for several test executions to be performed simultaneously on the same workstation, or even on different workstations inside the same network. The basic idea is that a central *test server* keeps track of the symbolic execution and the global progress of the testing process, and most of the practical work, including the concrete execution, is delegated to one or more *test clients*. Each client receives a path constraint corresponding to a test execution from the test server, solves it, and executes the program (previously instrumented for symbolic execution) with the solved input values. Instead of tracking the symbolic execution locally, the client communicates information on the encountered operations (and the associated concrete input values) back to the server. Once the concrete execution on the client terminates, the client notifies the server and also terminates. If the path constraint was unsatisfiable in the first place, a new path constraint is requested. The server keeps track of the symbolic execution by constructing an *execution tree* based on the messages received from the client. The execution tree consists of *execution nodes*, which represent the points in the program that are interesting in terms of the symbolic execution. This includes initializations, assignments and comparisons involving symbolic variables, synchronization points, branching points and so on. Each time a branch is encountered in the program, a corresponding branch is also created in the tree and the related symbolic constraints are assigned to the appropriate nodes. An unvisited leaf in the tree represents an uncovered execution path in the program, and the corresponding path constraint can be generated based on the constraints on the path from the leaf to the root of the tree. Once all the children of a node have been visited in a test execution, the node is considered finished and pruned from the tree to keep the size of the tree manageable.

The default *search order* or the order in which new execution paths are explored by the test server is basic *depth first search*. This means that the next node to be visited is always the node that was added to the tree most recently. The advantage of this method is that it keeps the tree relatively small as nodes can be pruned often, but the big downside is that is scales very poorly to large programs. The reason for this is that the testing process can spend a lot of time exploring a tiny branch of the program if the execution paths are very long, which can result in poor coverage of the program as a whole. Other search orders, such as *breadth first search* and *random search*, are also available, but they all are essentially just different trade-offs between the size of the tree (i.e. memory usage of the test server) and the provided coverage on the program.

One nice advantage of the server-client-architecture is that the server can be relatively agnostic of most of the implementation details of the client as long as the information it receives is correct and accurate in terms of the program under test. One corollary of this is that support for additional programming languages can be implemented mostly with changes on the client. The original LCT only supports automated testing of Java programs, but a client has also been implemented for C programs based on the LLVM [4, 37] framework. LLVM, originally *Low Level Virtual Machine*, is a sophisticated software compilation framework which provides a low-level source code representation and a set of tools that can be used to easily perform transformations on said representation. The compilation process with LLVM essentially consists of a *front-end* that transforms the program source code into this internal representation, a series of optimisations that are transformation passes on the internal representation, and a *back-end* that transforms it into executable machine code. The LCT instrumentation process for C programs, which adds additional statements to the program to allow the client-server-communication, is implemented as an LLVM optimisation pass.

3.3.2 Code Coverage

As our goal in terms of this work is to automatically generate a set of tests satisfying MC/DC on a program, it is beneficial to spend some time to consider the code coverage provided by LCT. Since LCT is based on DSE, DSE provides path coverage and path coverage seems far removed from MC/DC, this discussion might seem pointless, but when we take into account that LCT provides path coverage on the program *object code* (LLVM internal representation is essentially object code), the situation becomes more interesting. An important detail in MC/DC is that it is defined exclusively on the program source code, which makes sense when its application in the aviation software certification process is considered as discussed in Section 2.3. This means that instead path coverage and MC/DC, we should consider the relationship between path coverage on the object code and MC/DC, which are actually closer to each other than it initially seems.

The reason for this is the fact that during the compilation from source code to object code, the boolean expressions in branch conditions are broken down according to the *short-circuit semantics* of the programming language in question. With C this means that if a boolean expression is already known to be true or false, the correct branch should be taken immediately and the expression should not be evaluated further. In the object code representation of a program this means that each branch with a complicated boolean expression is separated into a series of branches with sub-expressions of the original expression that are only evaluated if necessary. This process can be seen in action in the running example provided in the next chapter (Section 4.3), which is not needlessly repeated here. When this is taken into account, complete path coverage of the branching structure on the object code level, which is something LCT provides, actually covers most of the requirements of MC/DC (Section 2.2) directly. Namely all entry and exit points in the program are covered (i) and all conditions (ii) and decision (iii) evaluate to both true and false at some point during the testing process. The only requirements that is missing is the last one (iv), but even it is almost covered, as taking all paths through the branching structure ends up exercising many combinations of condition evaluations. However, unlike the first three, the fourth requirement is not something we can count on being covered, so the test generation process must still make sure that all requirements are satisfied completely. This observation has several implications on the implementation details of the coverage-based test generation process, which is presented in the next chapter.

Chapter 4

Coverage-Based Test Generation

In this chapter we present how our coverage-based test generation process, based on DSE with additional *goal constraints*, is designed and implemented. Section 4.1 first discusses the practical issues that should be taken into account when automated test generation for MC/DC is considered. After this, the framework of our solution is presented in Section 4.2, which defines what goal constraints actually are and describes how they can be used to implement a test selection process for DSE. Once the background has been established, the test generation process itself is presented through the use of a running example in Section 4.3, and finally the interesting implementation details are provided in Section 4.4.

4.1 MC/DC and Test Generation

Automatically generating a set of tests to provide MC/DC for a software system is not exactly straightforward, especially if applicability to the aviation software certification process is also taken into account. The very strong safety culture behind this certification process clearly dictates how MC/DC should be both interpreted and used in practice, and some of the details are particularly difficult for the automated test generation process to handle. Here we consider some of these problems and propose solutions to them accordingly. There are also a few issues that are less practical and more philosophical in nature associated with applying automated test generation to MC/DC, including the one discussed in Section 2.3 involving the reduced effectiveness of the testing process as a reliability metric if expertise is blindly replaced with automation. These kinds of problems do not generally have clear solutions, and as such are not discussed here. Instead, they are covered later in Chapter 6.

The biggest problem in automating the test generation process for MC/DC while taking the aviation software certification process into account is that the tests for a system should be generated from the requirements (or specification) of the system. Even though the requirements in this case should be quite detailed (they should have the same branching structure as the source code, see Section 2.3), automatically extracting a set of tests from an essentially arbitrary set of requirements is very difficult, even when not considering the fact that the tests should also provide a decent degree of code coverage on the implementation. This problem can be solved with a slight reinterpretation of the term "specification", and generating the tests based on the source code of the system instead. With this change, we can do coverage-driven test generation by taking an existing automated software testing tool (namely LCT, see Section 3.3), directing its testing process to provide MC/DC and implementing a way to extract the interesting test executions into a set of executable test cases. Even though this is pretty much the best we can do in the scope of this work, the whole idea does go very much against the intent of DO-178B, and the meanings of requirements and structural coverage for a system are diminished as a result. This is the primary reason why our methods are not a suitable replacement for the traditional testing process and should not be used as such.

One issue with this test generation method is that that LCT works on the object code of the program. This is problematic because we are specifically interested in the coverage of the source code, and the structure of the program can change considerably during the compilation process. A standard solution for handling this particular problem is covered in [25]: Even though MC/DC on object code level does not necessarily equal MC/DC on source code level, demonstrating MC/DC on the object code is sufficient if analysis showing the equivalence between the two levels of coverage is also provided. This means that in order to satisfy the testing process requirements, nontrivial analysis arguing why code coverage on object code level implies code coverage on source code level should be performed. Such analysis is fortunately not necessary in the case of our test generation process, as the part of the process directing the testing and measuring the code coverage can actually be implemented completely on source code level. This means that even though the underlying testing tool works on the object code, any MC/DC provided by the produced tests is proper code coverage on the source code.

As a final note, it should be exclaimed again that we do not give any guarantees on the completeness of the generated set of tests. There are several reasons for this, the most important being the fact that the testing tool works during the execution of the program. This alone limits our consideration to executions that actually exist, meaning that for instance the presence of dead code can not be shown with the test set. We can provide an estimate on the amount of MC/DC achieved during the testing process, but since coverage measurement is not the primary function of the testing tool, this value should be taken with a grain of salt.

4.2 Goal Constraints

The basis of our coverage-based test generation process is a new concept called goal constraints, or simply *goals*, which are essentially additional constraints on the values of variables that can be inserted directly into the program source code. It is up the testing process to decide how the goals are used, but in our case they are treated as requirements that state “this constraint should be satisfied at this point in the execution at least once during the testing process”. This means that the goals are thought of as suggestions rather than restrictions on the actual values of variables.

As was already hinted at in Section 3.3.2, plain DSE, especially when performed on the program object code, is not directly applicable to automated test generation for a specific coverage criterion because it is so closely tied to path coverage. In the case of MC/DC, a test set that provides complete path coverage has both too many and too little test cases. On one hand many of the test cases are unnecessary as coverage of all execution paths is not required by MC/DC, and on the other hand the independence effect of each condition in each decision is not necessarily shown. This means that in order to apply DSE to automated test generation for MC/DC, we need a way to filter out uninteresting test runs and to perform additional test runs as necessary. Both of these issues can be solved with goal constraints, as long as the complete structure of the desired coverage criterion can be expressed in terms of constraints in the program source code.

The goals are intended to be used as an additional layer on top the DSE process as follows: Every time a goal is encountered during a test execution, the associated constraint is evaluated with the current concrete input values. If the constraint is satisfied, we know that the execution is interesting in terms of the coverage criterion, and a test case is generated and included in the test set based on the input values of the execution. Since each goal is only required to be satisfied once during the testing process, the global state of satisfied goals has to be stored outside the individual test executions. This way goals that have already been satisfied by some earlier test execution can safely be ignored. If a goal is not satisfied naturally during the testing process, additional test executions are performed by combining the goal constraint with the path constraint of the goal statement. This makes sure that if the goal can be satisfied on a particular execution path, input values that both reach and satisfy it are selected, and a test case for it is generated. Together with the path coverage provided by DSE, this process guarantees that every reachable goal in the program that is satisfiable in the first place has a corresponding test case, and that not unnecessary test cases are included in the test set.

The process is made slightly more complicated by the fact that even though the goals are defined in terms of the program source code, the actual testing process still operates on the object code. This means that goals, which appear only once in the source code, may actually be encountered in several different contexts in the object code after the program has been compiled. This is a small problem because variables are only considered to be symbolic if they depend on input values, which means that the goal constraint can change depending on the execution path it appears at. The details of how the situation is handled are provided in Section 4.4.1, but the result is that each goal is also accompanied by a unique identifier which can be used to recognise different instances of the same goal during the testing process. These identifiers also have an additional benefit to them, as they can be used to provide a rudimentary measure of code coverage. Normally a goal that is outside the reachable part of the program is never executed and thus rendered completely invisible to the testing process. With the identifiers however, if the testing process is aware of the total number of goals in the program, the number of goals that are either satisfied, encountered but not satisfied, or not encountered at all can all be reported at the end of the testing process.

4.3 Test Generation Process

This section describes our coverage-based test generation process step by step with the help of a running example. The example program, available in Figure 4.1, is very similar to the first example program presented in Chapter 2. The program contains three input parameters (the initialisation of which is omitted to reduce the size of the example), and no error conditions. The lack of error conditions does not actually limit the meaningfulness of this example in any way, as our primary goal is not to find errors in the program but to generate a set of tests that provides MC/DC. The test generation process can be split into two distinct parts, the constraint generation and instrumentation (covered in Section 4.3.1) and the test execution (covered in Section 4.3.2).

```
bool test(x, y, z) {
    if (x > 0 || (y < 0 && z == 0)) {
        return true;
    }
    return false;
}
```

Figure 4.1: Test generation example

4.3.1 Constraint Generation

As a small reminder from Section 2.2, the four requirements of MC/DC are: all entry and exit points in the program must be covered (i), all conditions (ii) and decisions (iii) must take all possible outcomes, and each condition in each decision must be shown to independently affect the outcome of the decision (iv). The first requirement is easily covered by a goal with an empty (always satisfiable) constraint in each entry and exit point of the program. The situation is similar with requirements two and three, as two goals that require the value of the condition or decision to be true and false can be inserted before or after every condition or decision in the program source code. It turns out this is actually not even necessary in practice, as the constraints generated for the last requirement also guarantee that every condition and decision is both true and false at least once during the testing process. The fourth requirement is again the one that the most interesting, and also the one that requires the most work.

The easiest way to show the independence effect of each condition would be to simply add goals for all possible combinations of condition evaluations before each decision in the program. This approach is similar to multiple condition coverage, and it also suffers from the same downside, namely the extremely large number of test cases that would be required to satisfy all such goals. Instead, we opt for a more clever approach presented in [59], which involves the generation of *positive* and *negative* sets of constraints based on the structure of the decision. The intuition behind this approach is that the boolean expression in the decision can be analysed recursively, and depending on the operations it contains two constraints forming the independence pair for each condition can be obtained. This method works according to the masked MC/DC semantics, which means that the positive and negative constraints are sufficient to show the effect of the condition even though the values of the other conditions are not necessarily held constant. The constraint sets are constructed according to the following equations:

$$x^+ = \{x\} \tag{4.1}$$

$$x^- = \{\neg x\} \tag{4.2}$$

$$(\neg A)^+ = A^- \tag{4.3}$$

$$(\neg A)^- = A^+ \tag{4.4}$$

$$(A \wedge B)^+ = \{a \wedge B \mid a \in A^+\} \cup \{A \wedge b \mid b \in B^+\} \tag{4.5}$$

$$(A \wedge B)^- = \{a \wedge B \mid a \in A^-\} \cup \{A \wedge b \mid b \in B^-\} \tag{4.6}$$

$$(A \vee B)^+ = \{a \wedge \neg B \mid a \in A^+\} \cup \{\neg A \wedge b \mid b \in B^+\} \tag{4.7}$$

$$(A \vee B)^- = \{a \wedge \neg B \mid a \in A^-\} \cup \{\neg A \wedge b \mid b \in B^-\} \tag{4.8}$$

Two first equations state that the positive set of a single condition is simply the condition itself, and respectively the negative set is the negation of the condition. The rest of the equations show how all the basic logic operations, negation, conjunction and disjunction, are handled. The equations for conjunction and disjunction are more complicated, as they split the set into two parts (one for the left and one for the right sub-expression). In the case of the running example we have one decision, `(x > 0 || (y < 0 && z == 0)):`, which consists of three conditions: X: `x > 0`, Y: `y < 0` and Z: `z == 0`. The positive and negative sets for this decision are constructed as follows:

$$\begin{aligned}
(X \vee (Y \wedge Z))^+ &= \{a \wedge \neg(Y \wedge Z) \mid a \in X^+\} \cup \{\neg X \wedge b \mid b \in (Y \wedge Z)^+\} \\
&= \{X \wedge \neg(Y \wedge Z)\} \cup \{\neg X \wedge b \mid b \in \{c \wedge Z \mid c \in Y^+\} \cup \{Y \wedge d \mid d \in Z^+\}\} \\
&= \{X \wedge \neg(Y \wedge Z)\} \cup \{\neg X \wedge b \mid b \in \{Y \wedge Z\} \cup \{Y \wedge Z\}\} \\
&= \{X \wedge \neg(Y \wedge Z)\} \cup \{\neg X \wedge b \mid b \in \{Y \wedge Z\}\} \\
&= \{X \wedge \neg(Y \wedge Z)\} \cup \{\neg X \wedge (Y \wedge Z)\} \\
&= \{X \wedge \neg(Y \wedge Z), \neg X \wedge Y \wedge Z\}
\end{aligned}$$

$$\begin{aligned}
(X \vee (Y \wedge Z))^- &= \{a \wedge \neg(Y \wedge Z) \mid a \in X^-\} \cup \{\neg X \wedge b \mid b \in (Y \wedge Z)^-\} \\
&= \{\neg X \wedge \neg(Y \wedge Z)\} \cup \{\neg X \wedge b \mid b \in \{c \wedge Z \mid c \in Y^-\} \cup \{Y \wedge d \mid d \in Z^-\}\} \\
&= \{\neg X \wedge \neg(Y \wedge Z)\} \cup \{\neg X \wedge b \mid b \in \{\neg Y \wedge Z\} \cup \{Y \wedge \neg Z\}\} \\
&= \{\neg X \wedge \neg(Y \wedge Z)\} \cup \{\neg X \wedge b \mid b \in \{\neg Y \wedge Z, Y \wedge \neg Z\}\} \\
&= \{\neg X \wedge \neg(Y \wedge Z)\} \cup \{\neg X \wedge (\neg Y \wedge Z), \neg X \wedge (Y \wedge \neg Z)\} \\
&= \{\neg X \wedge \neg(Y \wedge Z), \neg X \wedge \neg Y \wedge Z, \neg X \wedge Y \wedge \neg Z\}
\end{aligned}$$

Notice that the positive set has one constraint less than the negative set. This follows from the fact that with this decision, the positive effect of the conditions Y and Z can be shown with the same constraint. When the sets are combined, the set $G = \{X \wedge \neg(Y \wedge Z), \neg X \wedge Y \wedge Z, \neg X \wedge \neg(Y \wedge Z), \neg X \wedge \neg Y \wedge Z, \neg X \wedge Y \wedge \neg Z\}$ is obtained. This is the set of goal constraints that are necessary to satisfy the fourth MC/DC requirement for this decision, and as the program only has one decision, also for this program. Each of these constraints is then assigned an identifier and inserted into the source code, which results in the program available in Figure 4.2.

```
bool test(x, y, z) {
    goal(0, x > 0 ∧ ¬(y < 0 ∧ z = 0));
    goal(1, ¬(x > 0) ∧ y < 0 ∧ z = 0);
    goal(2, ¬(x > 0) ∧ ¬(y < 0 ∧ z = 0));
    goal(3, ¬(x > 0) ∧ ¬(y < 0) ∧ z = 0);
    goal(4, ¬(x > 0) ∧ y < 0 ∧ ¬(z = 0));
    if (x > 0 || (y < 0 && z == 0)) {
        return true;
    }
    return false;
}
```

Figure 4.2: Test generation example with goals

4.3.2 Testing Process

Before the testing process begins the program is instrumented and compiled, which means that we move from source code to object code. For the running example this means that the structure of the boolean expression in the branch is broken down according to the short-circuit semantics as was discussed in Section 3.3.2. As the details of the actual LLVM internal representation are not the point of this example, the approximate effect is replicated with a slight rewrite of the program structure, available in Figure 4.3. Even though the effect of the instrumentation process, which enables the symbolic execution, is also omitted in the example, it should be noted that that the goal statements are mostly unaffected by this process. The details of how goals area instrumented are provided later in Section 4.4.1.

Goals are managed during the testing process according to a relatively simple strategy: Every time a goal is encountered during a test execution, it is handled as described in Section 4.2 (the constraint is evaluated and if it is satisfied a test case based on the execution is generated). However, if

```
bool test(x, y, z) {
    goal(0, x > 0 ∧ ¬(y < 0 ∧ z = 0));
    goal(1, ¬(x > 0) ∧ y < 0 ∧ z = 0);
    goal(2, ¬(x > 0) ∧ ¬(y < 0 ∧ z = 0));
    goal(3, ¬(x > 0) ∧ ¬(y < 0) ∧ z = 0);
    goal(4, ¬(x > 0) ∧ y < 0 ∧ ¬(z = 0));
    if (x > 0) {
        return true;
    }
    if (y < 0) {
        if (z == 0) {
            return true;
        }
        return false;
    }
    return false;
}
```

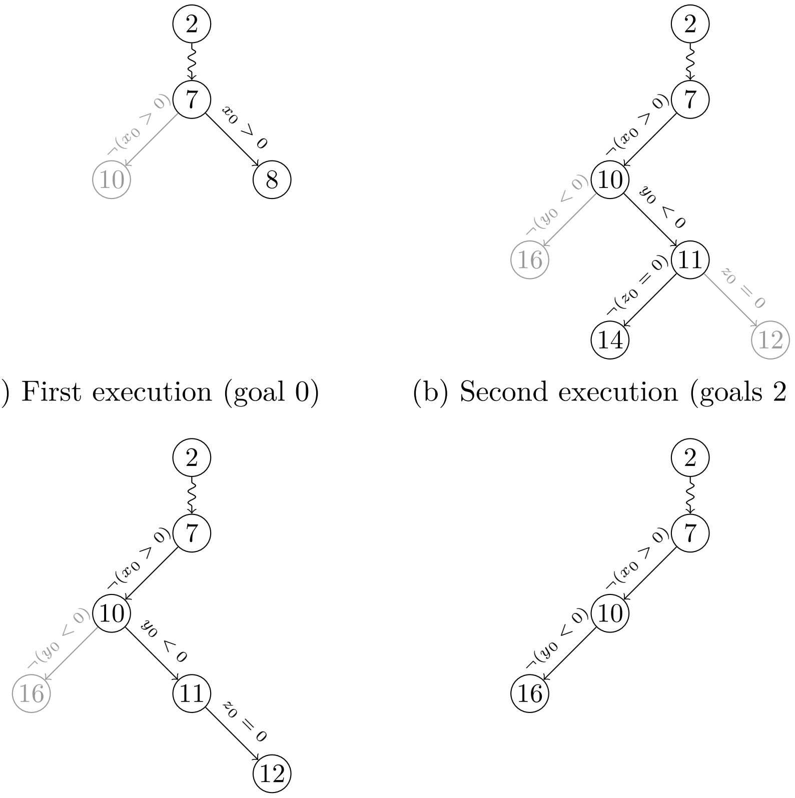
Figure 4.3: Test generation example with goals after compilation

the constraint is not satisfied, the goal is stored in the execution tree. More specifically, the goal is associated with the previous execution node on the same execution path. This is important because of the observation made in Section 3.3.2, which points out that the path coverage provided by DSE covers almost all of the MC/DC requirements essentially for free. This means that it is beneficial to only perform additional test runs to satisfy goals when absolutely necessary. Thus our strategy for coverage-based test generation is to proceed with the DSE process until a node which has unsatisfied goals associated with it is pruned from the tree. When this happens, the testing process attempts to satisfy each unsatisfied goal in the node before it is removed. If a goal can not be satisfied, it is not necessarily a problem since the same goal might still be satisfiable on some other execution path. A goal is reported as unsatisfiable only if it is still unsatisfied at the end of the testing process. With this strategy, we use minimal effort to make sure each goal is attempted to be satisfied on each execution path it appears at.

The progress of the goal augmented DSE process on the example program is provided in Figure 4.4. As DSE of a similar example program was already covered in Section 3.2, this example places a greater focus on the details of the test selection process. The only difference between these two examples in terms of DSE is that this case also includes the removal of completely explored execution nodes, since it is an important detail for the coverage-based test generation process for reasons explained above. Instead of listing the input value of each variable separately, the test executions are given as calls to the method `test(x,y,z)`, which can also be used to uniquely identify the generated test cases.

As nothing is known of the structure of the program before the first execution, the testing process starts with random input values. In this case the first execution (Figure 4.4a) happens to be `test(1,-1,-1)`. At the very start of the execution, the testing process notices the goal statements, which are evaluated and associated with the previous execution node in the execution tree. Since the tree is initially empty, the goals are stored in the root. The constraint in goal 0 is actually satisfiable with these input values, which means that the execution is marked as interesting and a new test case is generated at the end of the execution. The path taken by the first execution is 2, 7, 8, and as the node 8 does not have any unvisited children, it is pruned from the tree. This means that the path constraint for the second execution (Figure 4.4b) is constructed from the path 2, 7, 10, which can be satisfied with the input values `test(-1,-1,1)` for instance. This execution satisfies both goals 2 and 4, so a new test case is obtained. After the execution the node 14 can be pruned from the tree, but the node 11 can not as it still has one unexplored child, namely the node 12. The third execution (Figure 4.4c) attempts to reach that node, which is possible with the input values `test(-1,-1,0)`. A new test case is also generated based on this execution, since goal 1 is satisfied with these input values. Now the nodes 12 and 11 can be removed, which means that there is only one unexplored branch and only one unsatisfied goal left in the tree. The last branch, 2, 7, 10, 16, can be reached with the input values `test(-1,1,1)`. The fourth execution (Figure 4.4d) does not result in a new test case, since the input values do not satisfy the last goal constraint. As the remaining nodes are removed from the tree, the testing process notices that there is one unsatisfied goal left in the root. This means that an extra test execution (which is not shown in the figure) is performed, with input values obtained from the path constraint (which is empty) combined with the goal constraint. The input values `test(-1,1,0)` satisfy this constraint, which means that the goal is also satisfied during the execution and the last test case is obtained.

The resulting test set contains four test cases, namely `test(1,-1,-1)` (goal 0), `test(-1,-1,1)` (goals 2 and 4), `test(-1,-1,0)` (goal 1) and `test(-1,1,0)` (goal 3). It should be noted that the maximum number of generated test cases is equal to the number of goals in the source code, which is in turn equal to two times the number of conditions in the program. This is because a test case is only generated if at least one goal is satisfied during a test execution, and the goal constraint generation algorithm generates at most two constraints for each condition. In practice this number is usually lower, but it is still higher than the optimal $n + 1$ for n conditions obtained with maximal overlap between each independence pair (see Section 2.2). Some techniques for reducing this number further are discussed as topics for future research in Chapter 6.



(a) First execution (goal 0)

(b) Second execution (goals 2 and 4)

(c) Third execution (goal 1)

(d) Fourth execution (no goals)

Figure 4.4: Progress of the testing process

4.4 Implementation

The coverage-based test generation process is implemented as an extension to the automated software testing tool LCT (see Section 3.3). Like the process itself, the implementation can also be roughly split into two parts: the generation and instrumentation of goal constraints on the test client (Section 4.4.1), and the management of goals during the testing process on the test server (Section 4.4.2). Some changes are also made to the test case generation support in LCT, which are covered in Section 4.4.3. The limitations of the test generation process are discussed in Section 4.4.4. It should be noted again that even though LCT technically supports both Java and C programs, our solution is currently designed to only support C. The source code instrumentation techniques presented below could also be used to provide Java support, but as the test clients are completely different, this is left as a topic for future work. All software developed as a part of this work, including the contributions to LCT described in this section, are open source and released as a part of the LIME TestBench [3] toolkit.

4.4.1 Instrumentation

The goal generation and instrumentation into the program source code is done with a new tool called *mcdc-rewriter*, which is one of the main contributions of this work. The tool is written in the *C++* programming language [54], and it uses the utilities provided by the LLVM front-end *Clang* [1] to analyse and transform the program source code according to our needs. Clang is the LLVM front-end for C, C++, Objective C and Objective C++ languages, and it essentially performs the first steps of the compilation process by translating the program from source code to LLVM internal representation. The modular architecture of Clang is based on a set of libraries with open *application programming interfaces* (APIs), which make easy customisation of the compilation process and limited static analysis of the program source code possible.

The design of mcdc-rewriter is heavily inspired by a basic source-to-source transformation tool for C and C++ programs written by Eli Bendersky [7]. This translator is essentially a demonstration of the capabilities of the Clang API, and even though the transformation it provides is not especially interesting in itself, it is remarkably close to what is required of mcdc-rewriter in practice. The translator implements a custom visitor for the abstract syntax tree (AST) representation of the program generated during the compilation process, which adds a couple of comments around several types of interesting statements (specifically function definitions and if-statements) encountered in the AST. The AST of the program is a representation in which the structure of the program is laid out in tree form. The AST representation is especially convenient to us, as the boolean expressions in the program are also broken down according the short-circuit semantics of the programming language, which makes the constraint generation process significantly easier. What makes the AST modification process interesting is that Clang also provides a sophisticated *Rewriter* API for the AST, which tracks the changes and makes it possible to turn the modified AST back into valid source code with the changes still in place. To sum this up, the translator takes the program source code and, as the name suggests, automatically translates it into a copy with a set of comments added to certain places.

A similar approach can also be used to identify the statements in the program that are interesting in terms of MC/DC and to insert goal statements next to them as specified in Section 4.3.1. The step by step instrumentation process in mcdc-rewriter goes as follows: The AST representation of the program is first obtained by parsing the source code with Clang like in the normal compilation process. After this the AST is traversed by a custom recursive AST visitor, which recognises all the relevant points of interest in the program, specifically entry and exit points and boolean espressions, and generates appropriate goal constraints based on them. The AST is then carefully modified using the rewriting utilities in the Clang API in order to add the goal statements to their correct places. The same rewriting utilities are then used to produce a copy of the source code with the goal statements in place, which in turn can be processed by the test client like any other program. It should be noted that even though all entry and exit points are automatically covered during the testing process thanks to the properties of DSE, the always satisfiable goal statements associated with them are still useful as they guarantee that corresponding test cases are generated. After all the goals have been generated, one additional statement is added to the entry point of the program. This statement conveys the total number of goals to the test server, which allows us to accurately report statistics on the number of covered goals even if some of them are never encountered during the testing process.

In addition to the source code instrumentation, a separate instrumentation pass is performed on the program during the actual compilation process by the test client to make the communication with the test server and the symbolic execution possible. The only change the addition of goals imposes on this instrumentation process is that the constraints in the goal statements must be updated according to the state of variables on each execution path. This is important because the same goal can appear on many execution paths, and some variables that are symbolic on some execution paths are not necessarily symbolic on others. The problem here is that if a seemingly symbolic constraint containing non-symbolic variables is communicated to the test server, the test server is actually unable to know what to do with it, as it only keeps track of symbolic variables (even though the test server is aware of the concrete values of the symbolic variables, the non-symbolic variables are largely handled by the test client during the testing process). This issue is solved by replacing each occurrence of each variable that is not symbolic in each goal constraint with its concrete counterpart, which means that instead of the variable, the constraint communicated to the test server contains the value of the variable. This results in a setting where each variable the testing process can not affect is always evaluated according to its actual value on each execution path, which is in line with the intent of the DSE process.

4.4.2 Goal Management

Each time a goal is encountered during a test execution, the test client sends a message that contains the goal identifier and constraint to the test server. After this the constraint is evaluated on the test client, and another message is sent if the constraint was satisfied. The test server keeps track of the goals in the execution tree by storing them in the previous execution node on the same execution path, with each execution node able to contain an unspecified number of goals. In order to track the global state of which goals have already been satisfied and covered during the testing process efficiently, a separate *goal tracker* is generated and stored in a manner similar to the execution tree. The goal tracker contains two bit fields with one bit reserved for each goal (the goal identifiers are consecutive integers starting from 0), and they are initialised at the start of the testing process according to the message received from the test client which contains the total number of goals. It should be noted that even though the test client sends this message at the start of every test execution (as the test client is unaware of the global state of the testing process), the goal tracker is only initialised once and the message is ignored on each subsequent test execution. Each bit in the bit fields is initially zero, and is set to one when the goal is satisfied or covered. Every time the test server receives a message regarding a goal, it first checks the goal tracker to find out if the goal has already been satisfied. If this is the case, the message is ignored.

One potential downfall of storing the goals in the execution nodes is that it might lead to an infinite loop in the testing process. A situation might arise where the test server notices an unsatisfied goal in a node that is about to be pruned from the tree and attempts to satisfy it, which leads the next test execution down the same path and adds the same goal back to the node. In order to avoid this undesired situation, the goals are stored in the execution nodes in two separate lists: one for goals that have been added to the node at some point during the testing process, and one for goals that have been added but have not been attempted yet. When an unsatisfied goal is encountered during the testing process, it is added to both lists, and when a test execution to satisfy a goal is performed, the goal is removed from the second list. If a goal can be found in the first list, it is not added to the node again. This guarantees that each goal is only attempted once in each execution node, and thus solves the aforementioned problem.

When a goal is satisfied during a test execution, the goal is marked as satisfied in the goal tracker and a new test case is generated (the details of this process are available below in Section 4.4.3). It should be noted that the test server does not walk through the whole execution tree to remove instances of the satisfied goal from the execution nodes, as it is sufficient to simply consult the goal tracker each time just before a test execution to satisfy a goal is attempted. Our strategy for satisfying goals was already largely covered in Section 4.3.2, but the short story is that extra test executions are only performed when a node with unsatisfied goals is about to be removed from the tree. When this happens the lists in the node are updated, and a new test execution is performed with the constraint obtained by combining the path constraint in the node with the goal constraint. If input values that satisfy the constraint can be solved, the following test execution will encounter the same goal on the same execution path and satisfy it. Even if this is not the case, the goal is still removed from the node and the testing process proceeds as usual. Not being able to satisfy a goal is not a problem, as the goal might still be satisfiable in some other node. One important detail regarding the extra test executions is that, unlike with normal symbolic executions of the program, the execution tree is not updated during them. Even though this behaviour could sometimes be desired, as the addition of goal constraints to path constraints might lead the execution down a previously unexplored path and thus save time in terms of the whole testing process, we decided against it to make the goals as nondisruptive as possible for normal LCT operation, which allows the goals to work nicely with most of the search strategies supported by LCT. Some ideas on the possible benefits of using goals as a part of the search strategy are discussed in Chapter 6.

As the test server supports multiple concurrent test clients, the execution tree, the goal tracker and the goal lists in the execution nodes can all be updated simultaneously by different test executions. The underlying data structures in the case of the goal tracker and the goal lists are not inherently thread safe, so all access to them is synchronized in order to keep the data consistent. The goal lists are also initialised lazily (they are created just before they are accessed the first time) to avoid unnecessary memory usage, which is efficient because the goals are relatively clustered in the program source code, and thus the majority of execution nodes do not actually contain any goals during the testing process.

4.4.3 Test Case Generation

The last piece of the coverage-based test generation process is the generation of the test cases themselves. As our goal is to develop a test generation process which is practically useful in the context of the aviation software certification process (which is the main application of MC/DC), the produced test set should be independent in the sense that it can be executed without the test server. This is necessary to make the test cases comply nicely with different code coverage measurement tools, as the accurate measurement of code coverage often involves sophisticated dynamic analysis of the program, which is not necessarily compatible with the symbolic execution provided by the test server. LCT has rudimentary support for repeating the testing process without the test server by generating a set of unit tests based on the input values solved during the testing process. This is done by saving the input values into a special *test input file*, which can be parsed with a separate tool to generate a set of *JUnit* [2, 21] test cases. The downside of this process is that the JUnit framework is exclusive to Java programs, which means that the same test case generation is not directly applicable to C programs.

In terms of this work, the test case generation support in LCT is extended to also cover C programs. The idea behind the new test case generation process is largely similar to the one described above, with the exception that instead of relying on a dedicated unit testing framework, the program is compiled against a special library that provides a custom test client implementation, which reads the input values directly from the test input file instead of solving them based on the constraints obtained from the test server. This makes it possible to execute the tests executions performed during the testing process without the overhead of the client-server-communication, and also allows us to choose which test cases are included in the test set by filtering them on the test server during the actual testing process. When LCT is configured to generate a MC/DC test set, the input values are written to the test input file only if the test execution satisfies some previously unsatisfied goal. This is done by keeping track of which goals are satisfied during each test execution on the test server. This also allows us to augment the file with additional information containing the identifiers of the goals that are the reason each test case is included in the test set, as well as the identifiers of all other goals satisfied by each test case. The information is currently not used for anything, as we feel that including at most one test case for each goal in the program source code should provide a sufficiently concise test set, but it could be used to reduce the size of the test set even further by identifying and removing redundant test cases. One additional improvement to the test input file generation process is an optional configuration option, which makes the test server generate an independent test input file for each test case. This is mostly a practical improvement, which makes the execution of individual test cases slightly easier in the case the repetition of the entire testing process is not desired.

As the coverage-based test generation process is implemented as an optional feature on top of LCT, the test server can also be configured to ignore the goals either completely or partially during the testing process. This means that even if the program source code includes goal statements, the program can still be tested like any other program with LCT without the need to make additional changes to the source code. If the goals are ignored partially, the test server operates in a mode in which it tracks the goals as usual, but does not attempt to satisfy them with extra test executions. This allows us to use the goals as a basic coverage measurement tool for the normal LCT testing process to find out how much the goal management improves the size and quality of the generated test set. This functionality does not have many practical uses, but it is a central part of our tool evaluation, which is presented in the next chapter.

4.4.4 Limitations and Assumptions

The final thing that should be considered before the implementation evaluation is presented is the limitations of our test generation process. In the current state of the tool, the class of programs it supports is relatively small. This is mostly a result of the limited support for the C programming language in LCT, which means that for instance floating point numbers, arrays and dynamic memory allocation are not completely supported during the symbolic execution. Besides these limitations, there is one very important detail in the way goals are handled during the testing process that should be taken into account if our tool is to be used in practice. Recall that every time a goal is encountered during the testing process, the associated goal constraint is evaluated on the test client to find out if the goal is satisfied or not. As the goal constraint is obtained from a boolean expression in the program source code, this means that the boolean expression (or strictly speaking, the part of the boolean expression that is used to form the goal constraint) is evaluated twice during the testing process: once when the goal constraints is evaluated, and once when the expression itself is evaluated. This means that if the expression contains side effects, the side effects also happen twice. There are no immediately obvious workarounds to this issue, so we just have to assume the program does not contain boolean expressions with side effects or otherwise the test generation process might behave unexpectedly.

The constraint generation process in mcdc-rewriter is also somewhat limited and not thoroughly tested. The biggest limitation in the current implementation is that the only branching construction it supports is the if-statement. Most notably, switch-case-statements are not handled by the tool, even though they might also contain boolean expressions that are relevant in terms of MC/DC. The reason for this is that there is no clear direction to how switch-case-statements should be handled, as MC/DC test generation should generally be performed based on the specification of the system rather than the implementation. Fortunately switch-case-statements can always be rewritten as a series of if-statements, so one way to work around this limitation is to do exactly that. This process could also technically be automated, but it has not been implemented in the current version of mcdc-rewriter.

Chapter 5

Evaluation

This chapter presents an evaluation of our coverage-based test generation tool on a set of test programs. The experimental setup is described in Section 5.1, after which the results of the experiments are presented and analysed in Section 5.2.

5.1 Experiments

As our primary intention is to provide a tool that can be actually useful in the context of aviation software development, the performance of the test generation process, as long as it is sufficient, is largely secondary in importance to the quality of the generated test set. In the ideal situation, this evaluation should thus be able to show that our tool (and the underlying coverage-based test generation method) is reasonably *sound* and *complete*. This means that the tool should report complete MC/DC only if the generated test set actually provides it, and conversely that if such a test set exists for a program, the tool should be able to generate it. If we assume that the goal constraints instrumented in the program represent the MC/DC requirements accurately, we are able to find out if our tool implementation is sound and complete rather easily based on the statistics on satisfied and unsatisfied goals it reports. However, this is not sufficient to guarantee that the test generation process satisfies these properties, as the initial assumption might not necessarily be correct. So, in order to determine that the test generation process itself is sound and complete, we must resort to a separate coverage measurement tool to find out the actual MC/DC provided by the generated test set and compare it with the measured goal coverage. The good news is that there are tools, such as the *Rapita Verification Suite* [5], that are able to measure MC/DC and are sufficiently reliable to be used in practice, but the bad news is that these tools are generally not freely available. This is actually not too surprising if the expenses required to develop and verify such a tool are considered, but it also means that proper evaluation of our test generation tool is not possible in terms of this work. Due to these practical issues, this evaluation focuses instead on the other benefits the goal extension provides on top of the regular LCT testing process.

Since the current implementation of our test generation tool is rather limited as discussed in Section 4.4.4, finding suitable test programs that are complicated enough to be interesting in terms of MC/DC, simple enough to conform to these limitations and freely available proved to be difficult. In order to demonstrate the effect of the goal extension on a reasonably wide range of traditional program archetypes, this evaluation is performed on a set of five test programs specifically engineered with the limitations of the test generation process in mind. These test programs are included in the LIME TestBench [3] toolkit, and for the time being also available separately [6]. Each test program (augmented with a test harness that takes care of the initialisation of the input values) is first instrumented with goal constraints, after which two sets of tests are generated for it with LCT in two different modes: In the first mode LCT simply uses goals as a coverage measurement device as described in Section 4.4.3, and in the second mode additional test executions to satisfy goals are performed and the test set is also filtered based on goal coverage.

The first test program, mcdc-example, is basically identical to the example program in Figure 4.1 of Section 4.3. This program is not very practical as it does not do anything interesting by itself, but it is included in the experiments to make sure the test generation process works as expected. The second test program, mcdc-expression, is essentially a slightly more complicated variant of the first test program. This program takes five input values, and checks if they form a series in which each value is the product of its predecessors. The purpose of this test is to find out if there are practical cases where the object code path coverage provided by LCT is significantly different from the source code MC/DC provided by the extended test generation process. The third test program, mcdc-search, is an implementation of the *golden section search* [35] algorithm for integer-valued functions. The golden section search is a method for finding a minimum or maximum of a function inside a given range, provided that the function is strictly *unimodal* (has only one minimum or maximum) inside that range. The algorithm is similar to *Fibonacci search*, and it gets its name from the use of the golden section to guarantee that the search interval shrinks by a constant factor on each iteration. In our test program, input values are used as factors for a second degree polynomial which is minimised inside a fixed range. The last two test programs, mcdc-search-5 and mcdc-search-7, are traditional in-place *quicksort* implementations for an array of integers. The size of the array which is sorted is 5 and 7 respectively, and the contents of the array are initialised with input values. These last three test programs are examples of traditional program archetypes, and are included in the experiments to find out what are the expected effects of the goal extension in practical applications.

The experiments are performed on a workstation with 8GB of RAM and a 3.30GHz Intel Core i5-2500 CPU with four cores and 6MB of L3 cache. The CPU usage is limited to only one core in the experiments, since the test generation process is not configured to utilise the distributed architecture of LCT to its full potential and is instead performed one test execution at a time. Each experiment measures the amount of CPU time in seconds required to generate the test set, the size of the test set and the number of goals satisfied and encountered during the test generation process. An estimate of the MC/DC provided by the test generation process is also reported as a percentage based on the number of goals that are satisfied from the total number of goals included in the test program source code. As the default search strategy of LCT is able to prune the execution tree efficiently and thus keep the size of the execution tree almost constant, we decided to not include the memory usage of the test generation process in these measurements. It should also be noted that since the workstation is included in a pool of computational resource available to the department, some variance might be introduced to the measured CPU time based on increased load on the workstation at certain times. To combat this, each experiment was repeated ten times and the reported CPU time is the average of these repetitions.

5.2 Results

The results of the experiments are presented in Table 5.1. Before each experiment is examined more closely, some initial observations can immediately be made from the results as a whole. It is evident that the object code path coverage provided by LCT is in most cases sufficient to provide complete goal coverage, unless extremely complicated boolean expression are included in the program. This is actually very much in line with the initial observation we made in Section 3.3.2, but it also indicates that the practical benefits of MC/DC over path coverage are somewhat limited if the program only contains decisions with few conditions. It seems that the CPU times between the experiments with and without goals are largely similar, with only small overhead caused by the additional test executions in some cases. This also relates most likely to the close relationship between object code path coverage

Program	**Time**	**Tests**	**Goals**			
			Satisfied	Found	Total	Coverage
mcdc-example	0.1216	4	8	8	8	100.0
with goals	0.1184	4	8	8	8	100.0
mcdc-expression	2.0636	17	15	22	22	68.18
with goals	2.5364	11	19	22	22	86.36
mcdc-search	18.5982	33	20	20	20	100.0
with goals	18.9266	3	20	20	20	100.0
mcdc-sort-5	4.9336	120	6	6	6	100.0
with goals	4.9436	1	6	6	6	100.0
mcdc-sort-7	225.6868	5040	6	6	6	100.0
with goals	226.3736	1	6	6	6	100.0

Table 5.1: Results of the experiments

and source code MC/DC, which means that additional test executions to satisfy goals are rarely necessary as most of the goals are already satisfied during the DSE process. This also supports our decision to only attempt to solve goals at the last possible moment rather well. It should also be noted that none of the test programs seems to have unreachable goals, as the number of found goals matches the total number of goals in all experiments. This is not surprising either, as the presence of unreachable goals would indicate that a part of the test program is beyond the reach of the DSE process, which is not the case with any of the test programs.

For the mcdc-example test program, the results are pretty much as expected. The only curious thing is that the test generation process was able to reach full goal coverage even when it did not attempt to explicitly satisfy goals, which did not happen in the example presented in Section 4.3.2. This disparity can be explained with the fact that the input values are chosen essentially randomly when they are not constrained, and here the values were chosen in such a way that all the goals were satisfied during the initial four test executions. This test programs also contains three additional goals that were not present in the example. These goals are the always satisfiable goals associated with the various entry and exit points of the program, that were not included in the example for the purpose of clarity.

In the case of mcdc-expression, the differences in CPU time and goal coverage are more noticeable. It seems that programs such as this, ones that contain complicated boolean expression, are the ideal candidates to demonstrate the difference between path coverage and MC/DC. The additional test executions were able to satisfy four additional goals, which results in a significant increase in the MC/DC provided by the test set. This experiment also shows that our test selection method is working as expected, as the higher goal coverage could be reached with a reduced number of test cases. The fact that all the goals in the program could not be satisfied indicates either that something is wrong in the implementation of the test generation tool, or that complete MC/DC is simply not possible for this program. If the latter is true, then the program contains decisions with conditions that can not affect the outcome of the decision by themselves, which is quite possible.

The results of the last three experiments are largely similar and can be analysed together. The common theme in these experiments is that the LCT testing process is able to easily provide complete goal coverage with and without additional test executions, but the amount of regular test executions necessary to provide path coverage for the test program is extremely high. This results from the fact that even though these programs do not contain any complicated boolean expressions, they still contain multiple branches and very many possible execution paths. This is especially apparent with mcdc-sort-5 and mcdc-sort-7, where the addition of two elements into the array resulted in a substantial increase in the number of test cases for the regular LCT testing process. However, the important observation to make is that the number of test cases requires to satisfy all goals in the program source code is extremely small in each of these experiments. The reason for this is that even though complete path coverage is likely to also provide MC/DC as a side effect, the relationship does not hold in the other direction: A test set that provides MC/DC is not likely to provide path coverage, and similarly the vast majority of test cases in the test set that provides path coverage are completely unnecessary in terms of MC/DC. In these experiments, it turns out complete goal coverage is already provided by the very first test executions, which renders most of the testing process unnecessary if goal coverage is the only desired outcome. This means that our test generation process could be modified slightly to terminate the testing process when complete goal coverage has been achieved, which could very likely translate to a significant reduction in the CPU-time required to generate the test set for a certain class of programs.

Chapter 6

Conclusions and Future Work

This work presented an automated MC/DC-based test generation process for C programs based on DSE with additional goal constraints in the program source code. An existing software testing tool called LCT was extended with goal support, and a completely new tool was developed to automatically generate and instrument a set of goals into the source code of a program. The implementation was evaluated on a set of test programs, and the results showed a significant reduction in the size of the test set and a slight improvement in code coverage for certain types of programs, with only minimal overhead in performance.

The main contribution of this work is the concept of goal constraints, which is a novel approach to test case selection and generation based on the extensive code coverage provided by DSE. We already showed how a demanding coverage criterion such as MC/DC can be expressed in terms of goal constraints, but the real versatility of this method was left largely unexplored. What we did not previously emphasise is that most of the commonly used coverage criteria are relatively easily defined in terms of constraints on the values of variables in the program source code, which means that the same goal-based DSE process can be used to generate test sets for a wide variety of different applications. From the coverage criteria presented in Section 2.1, condition/decision coverage is expressible with a subset of the set of constraints generated for MC/DC, and branch and statement coverage can be expressed with always satisfiable constraints in each branch and next to each executable statement in the program respectively. Path coverage is slightly more complicated, but it is also slightly less interesting as it is already provided directly by plain DSE. If explicit path coverage on the source code and not on the object code is required, goal constraints for it can be constructed in a manner similar to how path constraints are constructed during the DSE process, even though this requires significantly more work.

Goal constraints can also prove to be useful beyond their intended application as a test selection mechanism, which became apparent with the coverage measurement process described in Section 4.4.3 and used in our tool evaluation. Even though this particular process is not extremely interesting, the underlying idea could be extended to provide a dedicated coverage measurement tool for a separately created test set by simply evaluating the goal constraints during the testing process and observing how many of them are satisfied. There are also other approaches to goal constraints that could prove to be interesting topics for future research, such as the integration of goals into the search strategy of LCT. As goal constraints essentially describe combinations of variable values that are interesting by design, a search strategy built around the idea of satisfying goals as aggressively as possible could prove to be better at directing the testing process to cover interesting program behaviour than a simple search algorithm. Even though this would mean that most of the benefits gained from the current nondisruptive approach described in Section 4.4.2 would be lost, the improved coverage could prove to be a worthy trade, especially for very large programs that are already problematic for DSE.

There are also some limitations in the implementation of our coverage-based test generation process that should be addressed in possible future work. The most pressing issue is that the quality of the generated test cases was not properly evaluated, since we did not have access to a dedicated coverage measurement tool that could accurately measure the MC/DC provided by a test set. This is definitely something that should be addressed, as it is technically possible that our test generation process is somehow inherently flawed and fails to provide reasonable code coverage in some case that was not considered in this work. Besides solving the limitations of the tool discussed in Section 4.4.4, some improvements could also be made to the test set generation as discussed in Section 4.4.3. Since our tool already provides information on which goals are satisfied by which test case, a separate filtration method for the produced test set, or even a more general improvement for the test selection process, could be implemented based on a number of methods studied in the literature [28]. Another obvious topic for future work is the extension of the coverage-based test generation process to other programming languages besides C. As the Clang front-end also supports the programming languages C++, Objective C and Objective C++, our tool should technically work with any of these with little to no modification. The limiting factor in the range of supported languages is likely LCT itself, as some of the features present in more advanced programming languages might not be supported by the symbolic execution process.

All in all, the coverage-based test generation process developed around the concept of goal constraints turns out to be a practical solution to an otherwise difficult problem. As a final thought, the implications of using automated test generation for MC/DC in the context of the aviation software certification process should be considered. Section 4.1 already covered most of the practical problems associated with this process, but the more difficult philosophical problems were left mostly unanswered. The common core of these issues is that even though MC/DC is decent at finding errors, it is still lacking compared to similar but slightly more demanding coverage criteria [61]. This idea is further enforced by a study on the effectiveness of the certification process [49], in which the author states that even though there is no official record of a software error being the primary cause of an aircraft crash up to date, this success can more likely be attributed to the extremely strong safety culture in aviation software development and less likely to the use of MC/DC specifically. Combined with the result that a significant number of errors found in safety-critical software systems in general are not caused by incorrect implementation but either incorrect or misunderstood specification [39], what follow is that the selected coverage criterion is far less important than the fact that a person has to take the time to actually look through the specification and generate the test cases. Automating the test generation process can thus have an adverse effect on the reliability of the software system, unless extreme care is taken to verify that the original intent of the testing process is still satisfied. For our tool this means that even though it can make the test generation process significantly less demanding on the software developers, using it in practice is not necessarily an inherently good idea. The conclusion is that the same way MC/DC should not be considered to be a magic bullet that solves all reliability issues, our tool should not be treated as a one-button solution to the entire testing process, but merely as a reasonable starting point.

Bibliography

[1] Clang: a C language family frontend for LLVM. `http://clang.llvm.org/`.

[2] JUnit framework. `http://junit.org/`.

[3] LIME TestBench. `http://www.tcs.hut.fi/Software/lime/`.

[4] The LLVM compiler infrastructure. `http://llvm.org/`.

[5] Rapita verification suite. `http://www.rapitasystems.com/products/RVS`.

[6] Evaluation test programs. `http://users.ics.aalto.fi/~jkauttio/testprograms.tar.gz`, 2013.

[7] Bendersky, E. Basic source-to-source transformer with CLANG. `http://eli.thegreenplace.net/2012/06/08/basic-source-to-source-transformation-with-clang/`, 2012.

[8] Bird, D. L., and Munoz, C. U. Automatic generation of random self-checking test cases. *IBM Systems Journal 22*, 3 (1983), 229–245.

[9] Brummayer, R., and Biere, A. Boolector: An efficient SMT solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems, 15th International Conference, TACAS* (2009), S. Kowalewski and A. Philippou, Eds., vol. 5505 of *Lecture Notes in Computer Science*, Springer, pp. 174–177.

[10] Burnim, J., and Sen, K. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering, ASE* (2008), IEEE, pp. 443–446.

[11] Cadar, C., Dunbar, D., and Engler, D. R. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design*

and Implementation, OSDI (2008), R. Draves and R. van Renesse, Eds., USENIX Association, pp. 209–224.

[12] CAST. Rationale for accepting masking MC/DC in certification projects. Tech. rep., Certification Authorities Software Team, 2001. CAST-6.

[13] CAST. What is a "decision" in application of modified condition/decision coverage (MC/DC) and decision coverage (DC)? Tech. rep., Certification Authorities Software Team, 2002. CAST-10.

[14] Chilenski, J. J. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. Tech. rep., FAA, Office of Aviation Research, 2001.

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

[16] Clarke, E. M., Grumberg, O., and Peled, D. *Model checking.* MIT Press, 1999.

[17] de Moura, L. M., and Bjørner, N. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS* (2008), C. R. Ramakrishnan and J. Rehof, Eds., vol. 4963 of *Lecture Notes in Computer Science*, Springer, pp. 337–340.

[18] Dowson, M. The Ariane 5 software failure. *ACM SIGSOFT Software Engineering Notes 22*, 2 (1997), 84.

[19] Dupuy, A., and Leveson, N. An empirical evaluation of the MC/DC coverage criterion on the HETE-2 satellite software. In *19th Digital Avionics Systems Conference, DASC* (2000), IEEE, pp. 1B6/1–1B6/7.

[20] Forrester, J. E., and Miller, B. P. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows System Symposium* (2000), pp. 59–68.

[21] Gamma, E., and Beck, K. JUnit: A cook's tour. *Java Report 4*, 5 (1999), 27–38.

[22] Godefroid, P., Klarlund, N., and Sen, K. DART: Directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI* (2005), V. Sarkar and M. W. Hall, Eds., ACM, pp. 213–223.

[23] Godefroid, P., Levin, M. Y., and Molnar, D. A. Automated whitebox fuzz testing. In *Network and Distributed System Security Symposium, NDSS* (2008), The Internet Society.

[24] Gosling, J., Joy, B., Steele, G., Bracha, G., and Buckley, A. *The Java Language Specification, Java SE 7 Edition.* Addison-Wesley Professional, 2013.

[25] Hayhurst, K. J., Veerhusen, D. S., Chilenski, J. J., and Rierson, L. K. A practical tutorial on modified condition/decision coverage. Tech. rep., NASA, Langley Research Center, 2001.

[26] Heimdahl, M. P. E., Rayadurgam, S., Visser, W., Devaraj, G., and Gao, J. Auto-generating test sequences using model checkers: A case study. In *Third International Workshop on Formal Approaches to Testing of Software, FATES* (2003), A. Petrenko and A. Ulrich, Eds., vol. 2931 of *Lecture Notes in Computer Science*, Springer, pp. 42–59.

[27] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2013.

[28] Jones, J. A., and Harrold, M. J. Test-suite reduction and prioritization for modified condition/decision coverage. *IEEE Trans. Software Eng. 29*, 3 (2003), 195–209.

[29] Kähkönen, K. Automated dynamic test gneeration for sequential Java programs. Master's thesis, Helsinki University of Technology, Department of Information and Computer Science, 2008.

[30] Kähkönen, K., Kindermann, R., Heljanko, K., and Niemelä, I. Experimental comparison of concolic and random testing for Java Card applets. In *17th International SPIN Workshop* (2010), J. van de Pol and M. Weber, Eds., vol. 6349 of *Lecture Notes in Computer Science*, Springer, pp. 22–39.

[31] Kähkönen, K., Lampinen, J., Heljanko, K., and Niemelä, I. The LIME interface specification language and runtime monitoring tool. In *Runtime Verification, 9th International Workshop, RV* (2009), S. Bensalem and D. Peled, Eds., vol. 5779 of *Lecture Notes in Computer Science*, Springer, pp. 93–100.

[32] Kähkönen, K., Launiainen, T., Saarikivi, O., Kauttio, J., Heljanko, K., and Niemelä, I. LCT: An open source concolic testing tool for Java programs. In *6th Workshop on Bytecode Semantics,*

Verification, Analysis and Transformation, BYTECODE (2011), Elsevier, pp. 75–80.

[33] Kähkönen, K., Saarikivi, O., and Heljanko, K. Using unfoldings in automated testing of multithreaded programs. In *IEEE/ACM International Conference on Automated Software Engineering, ASE* (2012), M. Goedicke, T. Menzies, and M. Saeki, Eds., ACM, pp. 150–159.

[34] Kernighan, B. W., and Ritchie, D. *The C Programming Language*, second ed. Prentice-Hall, 1988.

[35] Kiefer, J. Sequential minimax search for a maximum. *Proceedings of the American Mathematical Society 4*, 3 (1953), 502–506.

[36] King, J. C. Symbolic execution and program testing. *Commun. ACM 19*, 7 (1976), 385–394.

[37] Lattner, C., and Adve, V. S. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization, CGO* (2004), IEEE Computer Society, pp. 75–88.

[38] Leveson, N. G., and Turner, C. S. Investigation of the Therac-25 accidents. *IEEE Computer 26*, 7 (1993), 18–41.

[39] Lutz, R. R. Analyzing software requirements errors in safety-critical, embedded systems. In *IEEE International Symposium on Requirements Engineering, RE* (1993), IEEE, pp. 126–133.

[40] Miller, J. C., and Maloney, C. J. Systematic mistake analysis of digital computer programs. *Commun. ACM 6*, 2 (1963), 58–63.

[41] Myers, G. J., Sandler, C., and Badgett, T. *The Art of Software Testing*, third ed. John Wiley & Sons, 2011.

[42] Offutt, A. J., and Hayes, J. H. A semantic model of program faults. In *International Symposium on Software Testing and Analysis, ISSTA* (1996), pp. 195–200.

[43] Peled, D. *Software Reliability Methods*. Springer, 2001.

[44] Pressman, R. S. *Software Engineering: A Practitioner's Approach*, seventh ed. McGraw-Hill Higher Education, 2010.

[45] Rayadurgam, S., and Heimdahl, M. P. E. Coverage based test-case generation using model checkers. In *8th IEEE International Conference on Engineering of Computer-Based Systems, ECBS* (2001), IEEE Computer Society, pp. 83–91.

[46] Reeves, G., and Neilson, T. The mars rover spirit FLASH anomaly. In *Aerospace Conference* (2005), IEEE, pp. 4186–4199.

[47] RTCA/SC-167. *Software considerations in airborne systems and equipment certification.* RTCA, Inc., 1992. Document No. DO-178B.

[48] RTCA/SC-205. *Software considerations in airborne systems and equipment certification.* RTCA, Inc., 2012. Document No. DO-178C.

[49] Rushby, J. M. New challenges in certification for aircraft software. In *1th International Conference on Embedded Software, EMSOFT* (2011), S. Chakraborty, A. Jerraya, S. K. Baruah, and S. Fischmeister, Eds., ACM, pp. 211–218.

[50] Saarikivi, O., Kähkönen, K., and Heljanko, K. Improving dynamic partial order reductions for concolic testing. In *12th International Conference on Application of Concurrency to System Design, ACSD* (2012), J. Brandt and K. Heljanko, Eds., IEEE, pp. 132–141.

[51] Sen, K. *Scalable Automated Methods for Dynamic Program Analysis.* PhD thesis, University of Illinois, 2006.

[52] Sen, K., Marinov, D., and Agha, G. CUTE: a concolic unit testing engine for C. In *10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/SIGSOFT FSE* (2005), M. Wermelinger and H. Gall, Eds., ACM, pp. 263–272.

[53] Storey, N. *Safety Critical Computer Systems.* Addison-Wesley, 1996.

[54] Stroustrup, B. *The C++ programming language*, fourth ed. Addison-Wesley Professional, 2013.

[55] Tassey, G. The economic impacts of inadequate infrastructure for software testing. Tech. rep., National Institute of Standards and Technology, 2002.

[56] Thai, T. L., and Lam, H. Q. *.NET framework essentials*, third ed. O'Reilly, 2003.

[57] Tillmann, N., and de Halleux, J. Pex - white box test generation for .NET. In *Tests and Proofs, Second International Conference, TAP* (2008), B. Beckert and R. Hähnle, Eds., vol. 4966 of *Lecture Notes in Computer Science*, Springer, pp. 134–153.

[58] Visser, W., Pasareanu, C. S., and Khurshid, S. Test input generation with java PathFinder. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA* (2004), G. S. Avrunin and G. Rothermel, Eds., ACM, pp. 97–107.

[59] Whalen, M. W., Rajan, A., Heimdahl, M. P. E., and Miller, S. P. Coverage metrics for requirements-based testing. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA* (2006), L. L. Pollock and M. Pezzè, Eds., ACM, pp. 25–36.

[60] Xie, T., Tillmann, N., de Halleux, J., and Schulte, W. Fitness-guided path exploration in dynamic symbolic execution. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN* (2009), IEEE, pp. 359–368.

[61] Yu, Y. T., and Lau, M. F. A comparison of MC/DC, MUMCUT and several other coverage criteria for logical decisions. *Journal of Systems and Software 79*, 5 (2006), 577–590.