UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

Modified Condition/Decision Coverage based on jumps

Sprungbasierte Messung von Modified Condition/Decision Coverage

Masterarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Felix Dino Lange

ausgegeben und betreut von
Prof. Dr. Martin Leucker

mit Unterstützung von
Malte Schmitz

Lübeck, den 14. Februar 2018

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne die Benutzung anderer als der angegebenen Hilfsmittel selbständig verfasst habe; die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet.

(Felix Dino Lange)
Lübeck, den 14. Februar 2018

Abstract Modified condition/decision coverage (MC/DC) is a coverage criterion, that is required for the certification of safety-critical software-systems used in the avionics industry.

Measuring MC/DC is usually done by instrumenting the code, which is intrusive and especially problematic in resource-limited systems. This thesis introduces a novel approach, that makes it possible to measure MC/DC without instrumentations. The basic idea is that every condition in the source code is translated to a conditional jump in the object code. This makes it possible to reconstruct the assignments of the conditions by analyzing program traces and to evaluate the coverage afterwards.

One possibility is to record the program trace and afterwards analyze the trace to find out which conditional jumps were executed. This approach is limited, because the size of traces becomes unfeasible very quickly.

To overcome these limitations an online approach is introduced, that combines current research on online trace reconstruction with online monitoring of trace information.

Kurzfassung Modified condition/decision coverage (MC/DC, auch zu deutsch Modifizierte Bedingungs-/Entscheidungsüberdeckung) ist ein Coverage-Kriterium, das für die Zulassung von sicherheitskritischen Softwaresystemen in der Luftfahrtindustrie vorgeschrieben ist.

Die Messung von MC/DC wird üblicherweise durch Codeinstrumentierung realisiert, was eine Veränderung des Codes mit sich bringt und insbesondere bei Systemen mit limitierten Resourcen zu Problemen führen kann. Diese Thesis stellt einen neuartigen Ansatz vor, der es ermöglicht MC/DC ohne Eingriff in den Code zu messen. Die Grundidee ist dabei, dass jede Bedingung im Quellcode zu einem bedingten Sprung im Objektcode übersetzt wird. Dies ermöglicht es die Belegung von Bedingungen durch eine Analyse von Programmtraces zu rekonstruieren und anschließend die Coverage zu evaluieren.

Es ist möglich den Trace aufzuzeichnen und anschließend zu analysieren, um herauszufinden welche bedingte Sprünge ausgeführt worden sind. Dieser Ansatz ist jedoch limitiert, da das Ausmaß der Traces sehr schnell sehr groß werden kann.

Um diese Limitierung zu umgehen, wird ein Online-Ansatz vorgestellt, der aktuelle Forschung an Online-Trace-Rekonstruierung mit Online-Monitoring verbindet.

Contents

1 Introduction

We trust our lives on software-based systems on a daily basis in aviation, automotive, medicine or other safety-critical systems. Malfunction of this kind of software can potentially lead to accidents, including tremendous damage and potential loss of lives. In order to prevent disastrous events certification standards, for example the DO-178C in the domain of aerospace software systems, are used by certification authorities, like the Federal Aviation Administration (FAA) and the European Aviation Safety Agency (EASA), to approve safety-critical software and ensure that the software used in the systems follows certain software engineering standards.

The verification process is an important part of the software life cycle and is there to find errors that were introduced during the software development. The verification guidance provided by the DO-178C includes a combination of reviews, analyses and tests. While reviews and analyses assess the accuracy, completeness and verifiability of the outputs of each phase in the software life cycle, testing verifies that the behavior of a system or a system component satisfies specified requirements. Detecting an error is equivalent to a behavior of the system that does not meet the requirements.

DO-178C requires that structural coverage analysis is performed during the verification process mainly as a completion criterion for the testing effort and to find dead code. There are multiple coverage criteria, that can be fulfilled during the structural coverage analysis and the level of code coverage depends on the safety level, which is determined by the damage that malfunction of the software can cause. Modified condition/decision coverage (MC/DC) is the coverage criterion that is required for software with the highest safety level A. In order to fulfill MC/DC the independent effect of all conditions that can influence the outcome of a decision has to be shown. MC/DC is a complex criterion and the definition of the independent effect has lead to wide discussions about the applicability and usefulness as a criterion. Supporter of the criterion praise that the decisions are guaranteed to be thoroughly tested and that the minimum numbers of test cases grow linearly with the number of conditions. On the other hand critics question the high costs, that are consumed in generating the needed number of test cases, and the ability of the criterion to find errors [Bha07].

Structural coverage analysis is usually done by instrumenting the source code to observe information about the taken paths, executed statements and evaluated con-

ditions. Instrumenting, i.e. software logging output, decreases the performance of the code significantly and is highly intrusive. After the verification process is completed, the developers have to decide if they either can leave the instrumentations inside the finished code or remove the instrumentations. The first option is especially problematic in resource limited embedded systems, because the instrumentation uses valuable memory and processing power. For the second option, it has to be shown that the instrumentation did not modify the behavior and that removing the instrumentation does not affect the measured coverage. Automatic code instrumentation and breakpoint-based debugging features of modern CPUs provide faster solutions but they are intrusive nonetheless and especially problematic for multi-core CPUs as they can introduce or hide errors for example due to race conditions[DGH+17].

Modern microprocessors feature embedded trace units (ETU), that deliver runtime information to debug ports of the processor. Because the information contains all executed instructions, it is delivered in a highly compressed format, which has to be reconstructed in order to gather the sequence of instructions and jumps executed by the processor. ARM CoresSight is one example if this technology and is available in most current ARM processors (Cortex M, R and A)[ARM13]. Usually the information by the ETU has to be recorded and reconstructed offline after the execution, but there are novel approaches how this can be done online during runtime, which makes it possible to monitor an execution for a basically unlimited amount of time. Online reconstruction of trace data is a challenging task and currently under research and described in [DGH+17].

The goal of this thesis is to show under what conditions MC/DC can be measured with no or minimal instrumentation. Static analysis of the source and object code is used to find out the interesting conditional jumps that correspond to conditions in the source code. This information can be used to analyze recorded traces that can be gathered with modern technology like IntelPT[Rei13] and ARM DSTREAM [ARM17]. This offline approach has the disadvantage that it is only possible to record a few seconds of trace data, because of enormously high data rates.

In a second approach MC/DC is evaluated how traces can be analyzed with online monitors based on the temporal stream based language TeSSLa. They can be build for the evaluation of MC/DC by analyzing the decision's structures. This approach is promising to allow long term observation of a system's behavior without temporal limitations.

1.1 Outline

The remainder of this thesis is structured as follows:

Chapter 2 describes the context of MC/DC and it's part in the structural coverage analysis and the verification of avionics software.

Chapter 3 discusses the technical details of MC/DC in particular with short-circuit logic and shows why object branch coverage is generally not sufficient to show MC/DC.

Chapter 4 describes how static analysis can be used to find conditional jumps that correspond to conditions. The addresses of these and the addresses that are the target of these jumps are then declared as watchpoints. With these information the assignments of the conditions during the execution of the code can be reconstructed by analyzing the trace.

Chapter 5 explains how MC/DC can be measured non-intrusively by analyzing traces offline. The limitations of this approach lead to the online approach.

Chapter 6 contains a concept how MC/DC can be measured non-intrusively online and describes how online monitors for MC/DC can be specified in a stream-based language.

Chapter 7 contains details about the used technologies and how before introduced ideas have been implemented.

2 Structural Coverage Analysis & DO-178C

2.1 Verification of Safety-Critical Software

The DO-178C/ED-12C, *Software Considerations in Airborne Systems and Equipment Certification* is a document published by the RTCA, Inc. (Radio Technical Commission for Aeronautics), which is used by certification authorities to approve commercial software-based aerospace systems. Software verification, an integral part of the DO-178C-compliant certification process, is essential for the development of safety-critical software. DO-178C defines verification as "The evaluation of the outputs of a process to ensure correctness and consistency with respect to the inputs and standards provided to that process". One important part of the verification process is *testing*, which is described as "the process exercising a system or system component to verify that it satisfies specified requirements and to detect errors"[Rie13].

DO-178C defines an *error* as "with respect to software, a mistake in requirements, design, or code"[Rie13]. Because errors are defined with respect to requirements and code, the DO-178C focuses on *requirements-based testing* to in order to detect errors before they can become faults or failures[HVCR01]. The purpose of software testing is to uncover errors that were made during the development phases. During requirements-based testing all tests should be derived from the requirements and their execution should show that the requirement, it is derived from, is met and ensure that there is no unintended functionality. Software testing can only find errors, but can never be used to prove that no error exists, because the domain of possible inputs is too large to test and there are too many possible paths through the program[KF99]. Therefore an exit criterion is needed that helps to verify that the testing effort is completed, for example of the DO-178C that is *structural coverage analysis*.

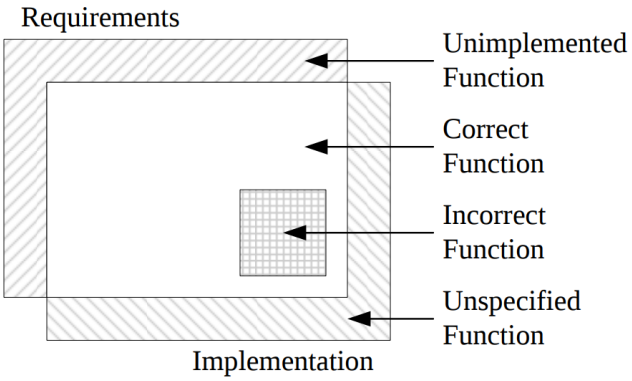


Figure 2.1: Requirements and Implementation Overlap[CK07]

2.2 Structural Coverage Analysis

Table A-7 in DO-178C is entitled "Verification of Verification Process Results" and requires evaluation of the adequacy and completeness of the requirements-based testing process by performing *requirements coverage analysis* and *structural coverage analysis*. While requirements coverage analysis determines which requirements have and have not been tested, structural coverage analysis has the following purposes[Rie13]:

- Ensures that all code has been executed at least once and that there is no dead code.
- Finds unintended and untested functionalities and helps identify incorrect logic.
- Serves as an objective completion criterion for the testing effort.

Figure 2.1 shows what kind of errors can be detected in the requirements-based testing process. Structural coverage analysis is generally capable of finding the functions in the code that were not specified in the requirements (*unspecified function*) and can help to find *incorrect functions*. To find the *unimplemented function* it is necessary to perform requirements coverage analysis, because they are described in the requirements but they are not implemented yet.

DO-178C requires different structural coverage criteria depending on the safety level of the software that is under verification. Depending on the effect that a failure of the software can have, it can be classified into five *Software Levels* or *Design Assurance Level*:

- Level A: catastrophic effect
- Level B: hazardous effect
- Level C: major effect
- Level D: minor effect
- Level E: no effect

In order to understand how MC/DC compares to other coverage criteria, the most common used criteria are introduced in the following. Generally speaking, the higher the software level, the more complicated the required coverage criteria become and the more test cases are needed to satisfy the criterion. The following brief definitions and explanations are based on the DO-178C or on references that are based on the DO-178C[Rie13].

2.2.1 Statement Coverage

Statement coverage is required for software levels A, B, and C. Achieving statement coverage shows that all code statements are reachable based on the test cases developed from the requirements. Due to the lack of detecting logical errors and the insensitivity to some control structures statement coverage is considered a relatively weak criterion[HVCR01].

Definition 2.1 (statement coverage)**.** DO-178C Table A-7 Objective 7 defines[Rie13]:

- "Statement coverage:
  - "Every statement in the program has been invoked at least one"

2.2.2 Decision Coverage

Decision coverage is required for software levels A and B. In context of the DO-178-C it must be noted that the commonly used definition that equals decision coverage with *branch* or *path coverage* must be extended with the literal definition of a decision in the DO-178C (see section 3.3 for details).

Definition 2.2 (decision coverage)**.** DO-178C Table A-7 Objective 6 defines[Rie13]:

- "Decision - A Boolean expression composed of conditions and zero or more Boolean operators. If a condition appears more than once in a decision, each occurrence is a distinct condition."

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

2.2.3 Multiple Condition Coverage

Multiple condition coverage guarantees to find every error that is caused by logical decisions, because every possible input combination has to be ensured in order to fulfilled the criterion. Because every combination (**True** and **False**) of every condition has to be tested, a minimum set of 2^n test cases needed for a decision with n conditions. This exponential growth can result in a test suite size that is not feasible and therefore the criterion is not required by the DO-178C.

Definition 2.3 (multiple condition coverage)**.** [Mye06] defines:

- "Multiple condition coverage:
  - "Every point of entry and exit in the program has been invoked at least once,
  - all possible combinations of the outcomes of the conditions within each decision have been taken at least once."

2.2.4 Modified Condition/Decision Coverage

Modified condition/decision coverage (MC/DC) is required for level A software. It was introduced to the aviation industry to address the concerns of testing complex Boolean expression. Less advanced criteria, like *decision coverage*, treat a decision as a single node in the program structure, regardless of its complexity[CM94].
MC/DC requires a minimum of *n+1* test cases for an expression with n uncoupled conditions[KC15]. This linear growth makes MC/DC more applicable than the *multiple conditions coverage*, which requires to test every possible combination of inputs in every decision and therefore has an exponential growth of test cases.

Definition 2.4 (modified condition/decision coverage)**.** DO-178C Table A-7 Objective 5[Rie13] defines:

- "Condition - A Boolean expression containing no Boolean operators except for the unary operator (NOT)."
- "Decision - A Boolean expression composed of conditions and zero or more Boolean operators. If a condition appears more than once in a decision, each occurrence is a distinct condition."
- "Modified condition/decision coverage:
  - Every point of entry and exit in the program has been invoked at least once,
  - every condition in a decision in the program has taken all possible outcomes at least once,
  - every decision in the program has taken all possible outcomes at least once, and
  - each condition in a decision has shown to independently affect that decision's outcome by: (1) varying just that condition while holding fixed all other possible conditions, or (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome."

The most important part of this definition is the demand to show the independent effect of conditions on their decision. See chapter 3 for the multiple interpretations of this definition and the technical details of MC/DC.

3 Technical Details of Modified Condition/Decision Coverage

The most challenging and in the literature ([Chi01],[CGHQ12],[Bha07]) most discussed part about fulfilling MC/DC is to show the independent effect of each condition in a decision. To show the independence there are at least two cases for each condition needed where only the outcome of the decision and the condition itself is toggled. These two cases are also called *independent pair*. Table 3.1 shows all input cases for an example decision $(A \vee (B \wedge C))$. For instance an independent pair for condition A are number 2 and 6 because only the value of A and the outcome of the decision is changing. Similarly, independent pairs for the other conditions can be found. The minimal sets of cases that show the independent effect of all conditions and therefore fulfill MC/DC are {2, 3, 4, 6} and {2, 3, 4, 7}.

	input:			decision:	pairs:		
Nr.	A	B	C	$A \vee (B \wedge C)$	A	B	C
1	F	F	F	F	5		
2	F	F	T	F	6	4	
3	F	T	F	F	7		4
4	F	T	T	T		2	3
5	T	F	F	T	1		
6	T	F	T	T	2		
7	T	T	F	T	3		
8	T	T	T	T			

Table 3.1: Unique-Cause MC/DC table with independent pairs for decision $A \vee (B \wedge C)$.

3.1 Multiple Interpretation of MC/DC

Other the last years the interpretation of what it means to show the independent effect of a condition has been interpreted in different ways. [Chi01] defines and extensively discusses three forms that are briefly introduced in the following.

Unique-Cause MC/DC is the strongest form of MC/DC and is the original interpretation of the DO-178B. It requires that a single condition and the decision's outcome is toggled to show that condition's independence. That definition implies that the values of all other conditions are known and that they are guaranteed to be fixed. If a decision contains strongly coupled conditions, it is not possible to show coverage. For example, it is not possible to show Unique-Cause MC/DC for the decision $A \vee (B \wedge \neg A)$, because the first condition A is strongly coupled to the third condition $\neg A$ and therefore it is not possible to change the value of one condition while holding the other fixed. That restriction makes Unique-Cause MC/DC not very useful in practice.

Unique-Cause + Masking MC/DC expands Unique-Cause MC/DC so that masking is allowed for strongly coupled conditions. Meaning that all conditions must be fixed excluding these that are strongly coupled. This definition complies with the interpretation of [CM94].

Masking MC/DC is the weakest form of MC/DC and allows masking in all cases. A condition is considered *masked* in this context, if varying this condition cannot affect the outcome of a decision. For example, it is sufficient to show the independent effect of A in $A \vee (B \wedge C)$ by holding the subexpression $B \wedge C$ fixed to **False** even if the values of B and C are changing. [Chi01] concludes that this should be the preferred form of MC/DC because, although Masking MC/DC allows less distinguishable test cases than Unique-Cause MC/DC, it's performance in detecting incorrect Boolean functions is not significantly different. Masking MC/DC is easier to satisfy because it allows for more independence pairs per condition.

In the position paper CAST-6 [T$^+$01] the Certification Authorities Software Team (CAST) compares Unique-Cause MC/DC and Masking MC/DC and concludes that Masking MC/DC meets the intent of the MC/DC objective and is therefore an acceptable method for meeting MC/DC with applicants striving to meet the objectives of DO-178B, Level A.

In DO-178C the definition of MC/DC was slightly modified and the amendment "... (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome."(see definition 2.2.4) was added[Pot12] in order to clarify the acceptance of Masking MC/DC.

```
1    0x40047d <+13>:    cmpl   $0x0,-0x8(%rbp)
2    0x400481 <+17>:    jne    0x40049b <testFunction+43>
3    0x400487 <+23>:    cmpl   $0x3,-0xc(%rbp)
4    0x40048b <+27>:    jge    0x4004a7 <testFunction+55>
5    0x400491 <+33>:    cmpl   $0x5,-0x10(%rbp)
6    0x400495 <+37>:    jne    0x4004a7 <testFunction+55>
7    0x40049b <+43>:    movl   $0x1,-0x4(%rbp)
8    0x4004a2 <+50>:    jmpq   0x4004ae <testFunction+62>
9    0x4004a7 <+55>:    movl   $0x0,-0x4(%rbp)
10   0x4004ae <+62>:    mov    -0x4(%rbp),%eax
11   0x4004b1 <+65>:    pop    %rbp
12   0x4004b2 <+66>:    retq
```

Figure 3.1: Object code for example decision A || (B && C).

3.2 MC/DC with Short-Circuit Logic

To evaluate MC/DC by analyzing program traces, that contain only information about the executed instructions and performed jumps, it is essential to consider how the compiler translates decisions into the object code. In most modern programming languages like C/C++ Boolean expressions are evaluated in strict order (left to right) and by using short circuit logic. For example, the right operand of the &&-operator is not evaluated if the left operand is **False** and right operand of the ||-operator is not evaluated if the left operand is **True**. In the remainder of this thesis the symbols && and || are used as the short-circuit operators, while $\wedge$ and $\vee$ stand for the traditional logical operators.

Figure 3.1 shows how the decision A || (B && C) is translated to object code in C using the Clang compiler. Note how every condition is translated into a conditional jump and short-circuit logic is used, if the target of a jump skips the evaluation of other conditions. For example, if the jump in line 2 is taken, the other conditional jumps in line 4 and 6 are not evaluated at all.

Table 3.2 shows the short-circuit behavior for all possible condition inputs. It can be seen that the cases 5, 6, 7 and 8 are not distinguishable by looking at the evaluated conditions because of the ||-operator. Cases 1 and 2 show the same behavior with respect to the &&-operator. This reduction in complexity will be utilized in the online MC/DC measurement process as described in chapter 6. Besides **True** and **False** a third value, e.g. **?**, is needed to express the situation that the condition has not been evaluated at all due to short-circuit evaluation.

To show the independent effect of each condition as is required in definition 2.2.4 there must be found at least two cases where only the value of this condition differs and also the value of the outcome of the decision differs (independent pairs). If condition were not evaluated at all (value of **?**) they are not accountable, meaning that their occurrence does not count as a different value of a condition. When

	input:			decision:	evaluated:		
Nr.	*A*	*B*	*C*	A || (B && C):	*A*	*B*	*C*
1	F	F	F	F	F	F	?
2	F	F	T	F	F	F	?
3	F	T	F	F	F	T	F
4	F	T	T	T	F	T	T
5	T	F	F	T	T	?	?
6	T	F	T	T	T	?	?
7	T	T	F	T	T	?	?
8	T	T	T	T	T	?	?

Table 3.2: Short-circuit evaluation for decision A || (B && C).

	evaluated:			decision:	pairs:		
Nr.	*A*	*B*	*C*	A || (B && C):	*A*	*B*	*C*
1	F	F	?	F	4	3	
2	F	T	F	F	4		4
3	F	T	T	T		1	
4	T	?	?	T	1,2		2

Table 3.3: All distinguishable cases and independent pairs for the decision A || (B && C).

the left-hand operand alone determines the outcome of the decision, the right-hand operand can be considered as *masked* in sense of *Masked MC/DC*[CGHQ12]. This assumption complies with the part of the DO-178C definition: "each condition in a decision has shown to independently affect that decision's outcome by: ... (2) varying just that condition while holding fixed all other possible conditions that could affect the outcome.", because a condition that is not evaluated cannot affect a decision's outcome.

For example, in table 3.3 cases 1 and 3 show independence for condition B with respect to Masking MC/DC, because the value of B and the outcome differs while the other conditions are not changing (A) or respectively were not evaluated at all (condition C, case 1). In this case all the possible distinguishable cases (1-4) are needed to show MC/DC of this decision.

3.3 Definition of a Decision

Because the term *decision* is commonly used synonymously with the term *branch point* the CAST position paper *CAST-10* was published in 2002 to clarify the meaning of decision in the context of the DO-178C. It states that MC/DC should apply to all decisions, not just those within a branch point. That means that additionally to the decision within a branch point all Boolean operations that appear (i.e. in assignment statements) have to be considered. This position prevents that level A and B software can be coded with all Boolean operators outside of components' code control constructs which would weaken the achieved test coverage significantly[T+02]. An example of this kind of cheating is provided in the following:

The decision:

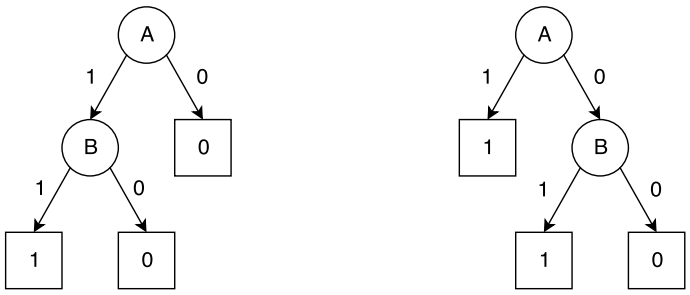
```
if((B || C) && D) then ...
```

can be expressed as:

```
A = B || C;
E = A && D;
if E then ...
```

In the first case at least four test cases are needed to show MC/DC while the second case would be covered by just assigning `E` to both **True** and **False** if a decision would be equal to a branch point. The CAST position paper argues that every logical structure should be thoroughly exercised, whether it occurs at a branch point or not.

This position provides two conclusions for this thesis. First the static analysis (section 4) has to account all logic structures, not just the branch points. Secondly logical structure, that maybe cannot be detected or are hardware-based, have to be evaluated externally.



(a) A && B **(b)** A || B

Figure 3.2: BDDs for decisions with two conditions with short-circuit evaluation.

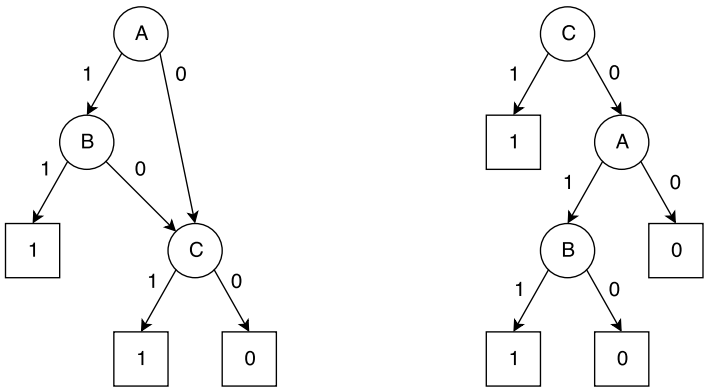
3.4 Relation to Object Branch Coverage

In earlier discussion papers published by the RTCA it was suggested that demonstrating Object Branch Coverage (OBC) implies MC/DC on source code level, if Boolean operators are restricted to short-circuit evaluation[CK07]. This assumption was thoroughly examined and discussed by Cyrille Comar et al. in an article called "Formalization and Comparison of MC/DC and Object Branch Coverage Criteria"[CGHQ12] and concluded not to be true in general.

Object Branch Coverage (OBC) is a coverage criterion that is defined on the object code level and is fulfilled, if all the branch instructions in the object code were taken both ways. A descriptive way to visualize the possible ways that can be taken in object code are *Binary Decision Diagrams* (BDDs), where every node in a BDD corresponds to a branch instruction in the object. By examining the BDDs that express certain decisions it can be shown that OBC does not imply MC/DC. In order to reduce the complexity and to narrow the problem short-circuit evaluation is assumed in the following.

To show the independent effect of the conditions in a decision with two conditions with short-circuit evaluation the three possible different executions equal the minimal set of cases that fulfill MC/DC. In figure 3.2 can be seen how these decisions are executed on object code level with three possible paths. These paths equal the possible distinguishable cases with short-circuit logic and therefore for decision with two conditions OBC is equivalent to MC/DC.

To show that OBC does not imply MC/DC the BDDs of decisions containing three or more conditions can be examined. As shown in figure 3.3a the BDDs of the decision (A && B) || C can be covered with only three paths. Because the minimal number of test cases needed for MC/DC is $n + 1$ the three cases that satisfy OBC cannot provide MC/DC, which needs at least four test cases. [CGHQ12] points out that there are certain decision there the implication (OBC $\Rightarrow$ MC/DC) holds depending



(a) (A && B) || C **(b)** C || (A && B)

Figure 3.3: BDDs for decisions with three conditions with short-circuit evaluation.

on the structure of the decision. In figure 3.3b the shown decision is very similar as the one in 3.3a, however the BDD of this decision needs four paths to fulfill OBC and these equal exactly those that are needed for MC/DC. In general the implication holds for decision with BDDs that are trees (with only one possible path from the root to any condition node). An equivalent property that characterizes cases where OBC implies MC/DC focusing on the logical structure of the decision is given in theorem Theorem 3.1 (*BDD branch coverage* equals OBC in that reference).

Theorem 3.1. *Given a decision D, BDD branch coverage implies MC/DC if, and only if, when considering the negation normal form D' of D, for every sub-decision E of D', all binary operators in the left-hand-side operand of E, if any, are of the same kind as E's operator[BCG+10].*

Because there is no commonly used coverage criterion defined directly on the object code level that is equivalent to MC/DC, it is necessary to reconstruct how the condition on source code level have been assigned during execution in order to measure MC/DC based on traces.

3.5 Coverage Analysis at the Object Code Level

Usually MC/DC is measured by instrumenting the source code to gather information about the execution and assignment of conditions and decisions. Because it is required by the DO-178C that the structural coverage analysis has to be performed on the code that is released, the instrumentation has to be left inside the airborne code. This is problematic because instrumentation consumes valuable resources. Additionally is it necessary to perform a source code to object code analysis to show that every line from the object code is traceable directly to the source code in order for compliance to DO-178C. If parts of the object code can't be traced back to the source code, then additional analysis must be provided[CAS03].

To overcome these problems MC/DC can be measured on object code level by analyzing program traces, which is explained in the following chapters. Additionally is structural coverage analysis desirable, because "it can support more 'valid' coverage as the testing and coverage analysis are conducted on an 'abstraction' of the code that is closer to the final airborne software to be installed than the source code"[CAS03].

The CAST-17 position paper "Structural Coverage of Object Code"[CAS03] provides certification authorities' concerns and position regarding the analysis of structural coverage at the object code level. It states that structural coverage analysis at the object code level can be proposed for compliance to the DO-178B, if the same level of assurance can be provided as the coverage analysis on the source code level.

One of those requirements is the traceability between the object code and the source code. In the following chapter is shown how static analysis can be used to trace the source code to object code and create a mapping between conditions and jumps in the object code. This information can then be used to declare which addresses of the object code are useful to reconstruct the condition on source code level.

4 Watchpoint Declaration by Static Analysis

To measure MC/DC based on the sequence of jumps performed by the CPU the source and the object code can be statically analyzed in order to find out which conditions in the source code correspond to which conditional jumps in the object code. This mapping is necessary because MC/DC is a criterion that is defined on the source code level and there are no equivalent metrics defined on the object code level. With this mapping the trace can be evaluated in order to reconstruct how the conditions were assigned during the execution. For demonstration purposes a simple code example (see figure 4.1a) is provided, on which the reconstruction process is explained. The code example shows an if statement, that contains a decision with three conditions. The conditions have different relational operators to show how they affect the object code.

4.1 Static Analysis of Source Code

The definition of MC/DC states that every condition in every decision in a program has to show it's independent effect. As stated in section 3.3 decisions in the source code include basic control statements that result in branch points in the program as well as other Boolean operators, that don't result in branch points, but they are decisions nevertheless.

The following program structures in case of the C language can contain decisions according to [T+02] and have to be detected:

- If statements
- Loops (while statements, do statements, for statements)
- Switch statements
- Assignment statements containing Boolean expressions that later are used in a branch point

In order to find such statements a preprocessing parser can be used to transform the source code in an abstract representation, which contains information about the statements, their location in the source code and how the statements are assembled. The C language compiler front-end *Clang* provides functionalities that transform C

code into an *Abstract Syntax Tree* (AST)[LA04]. In the following is explained how the AST can be analyzed to find if statements and assignment statements.

```
1  int testFunction(int a,int b,int c){
2       if(a || (b<3 && c==5)){
3         return 1;
4       }else{
5         return 0;
6       }
7  }
```

(a) Source code with if statement.

```
1  -IfStmt 0x55972de0b308 <line:2:6, line:6:5>
2   |-BinaryOperator 0x55972de0b230 <line:2:9, col:26> 'int' '||'
3   | | -ImplicitCastExpr 0x55972de0b218 <col:9> 'int' <LValueToRValue>
4   | |   -DeclRefExpr 0x55972de0b098 <col:9> 'int' lvalue ParmVar 0x55972ddb8720 'a' 'int'
5   |  -ParenExpr 0x55972de0b1f8 <col:14, col:26> 'int'
6   |   -BinaryOperator 0x55972de0b1d0 <col:15, col:25> 'int' '&&'
7   |     | -BinaryOperator 0x55972de0b120 <col:15, col:17> 'int' '<'
8   |     |   | -ImplicitCastExpr 0x55972de0b108 <col:15> 'int' <LValueToRValue>
9   |     |   |  -DeclRefExpr 0x55972de0b0c0 <col:15> 'int' lvalue ParmVar 0x55972ddb8798 'b' 'int'
10  |     |    -IntegerLiteral 0x55972de0b0e8 <col:17> 'int' 3
11  |      -BinaryOperator 0x55972de0b1a8 <col:22, col:25> 'int' '=='
12  |        | -ImplicitCastExpr 0x55972de0b190 <col:22> 'int' <LValueToRValue>
13  |        |  -DeclRefExpr 0x55972de0b148 <col:22> 'int' lvalue ParmVar 0x55972de0aeb8 'c' 'int'
14  |         -IntegerLiteral 0x55972de0b170 <col:25> 'int' 5
```

(b) Abstract Syntax Tree generated with clang.

condition:	relational operator:	line:	column:
a	none	2	9
$b < 3$	$<$	2	15
$c == 5$	$==$	2	22

(c) Substantial information obtained from the AST.

Figure 4.1: Example for an abstract representation of an if statement using Clang version 4.0.0.

4.1.1 If Statements

Finding an if statement in the AST is relatively simple, because there is a dedicated node for if statements. Loops and switch statements can be found in the same way.

The example code contains a decision (a || ($b < 3$ && $c == 5$)) in line 2 with the conditions a, $b < 3$ and $c == 5$. The corresponding AST can be seen in figure 4.1b. Note the "IfStmt" in line 1 and it's child-nodes containing all the information

needed like binary operators. As stated in section 4.2 it is essential to know the binary operators to correctly reconstruct the assignments based on the executed program jumps. Figure 4.1c shows what information can be obtained from the AST.

4.1.2 Assignment Statements

Note: The following approach has been theoretically discussed, but not practically implemented as part of this thesis.

In case of an assignment statement that contains a Boolean expression the detection is more complicated. They are only relevant as a decision, if they are later used in some form of branch point in the program. See figure 4.2 for an example.

Reconstructing the conditions is still possible, because the assignment in line 3 is translated into a conditional jump in the object code, that corresponds to the condition b. In line 5 the decision is evaluated by the if statement, which translates to another conditional jump. If b was **False**, then c was not evaluated at all. Otherwise equals a the value of c. With the conditional jump that evaluates the value of a in line 5, it is therefore possible to reconstruct the condition c indirectly.

Additionally, it is necessary to ensure that the value of the assignment has not changed between the initial statement and the branch point. Further static analysis can provide such insurance.

```
1  int testFunction(int b, int c){
2          int a;
3          a = b && c;
4
5          if(a){
6                  return 1;
7          }else{
8                  return 0;
9          }
10 }
```

Figure 4.2: Source code with assignment statement containing a Boolean expression.

4.2 Static Analysis of Object Code

In order to perform a reconstruction of the condition assignments it is required that every condition on the source code level translates to one specific conditional jump on the object code level. This assumption holds only if the compiler does not use any optimization level that influences conditional jumps. For example the gcc compiler uses on the first optimization level the options `-fif-conversion` and `-fif-conversion`, that transform conditional jumps into branch-less equivalents. Instead conditional moves, min, max, set flags and abs instructions are used[Tea]. It is not possible to detect how these kind of instructions are evaluated by analyzing traces, that only contain the sequence of executed instructions.

Assuming that every condition translates into exactly one conditional jump, their location in the object code has to be found. Most modern compilers provide features that emit debugging information e.g. in the DWARF format, that help developers to debug their code. Besides variable names and data structures it contains information about the origin of each line of the object code[E+07]. The static analysis of the source code in section 4.1 gives the location (line, column) of all decisions and conditions in the source code. By looking into the table (see figure 4.3) contained in the DWARF debugging format it is possible to map the conditions of the source directly to conditional jumps in the object code.

The before used example of the decision $(a \mid\mid (b < 3 \;\&\&\; c == 5))$ is continued in figure 4.3 and shows the mapping of each program line of the object code and the corresponding lines and columns in the source code from figure 4.1a. Because the location of the conditions in the source code is known from the analysis of the AST, it is now possible to map the conditions to addresses and conditional jumps in the object code. For example the condition $b < 3$ that is located at line 2, column 15 can be mapped to the object code address $0x400487$. Note in figure 4.4 that this address contains the compare-instruction `cmpl` and only the following instruction is the conditional jump.

Additionally it is necessary to know what kind of conditional jumps are used in the object code, because there is the possibility that a condition can be translated into it's negation. In that case a jump has to be interpreted that the condition was evaluated as **False**. For example the clang compiler with the x86-64 instruction set compiles conditions that contains the relational operator >= into a conditional jump with either the `jge` or the `jl` instruction. A detected jump in the first case (`jge`) means that the condition has been evaluated as **True** while a detected jump in the second case (`jl`) means that the condition has been evaluated as **False**. This behavior makes it necessary to detect the relational operator on the source code level as well as the instruction used on the object code level to correctly reconstruct the assignments by analyzing the executed jumps. The common conditional jump of

the x86-64 and the ARM architecture and their context dependent interpretations can be seen in table 4.1. These rules are strongly compiler-dependent and only have been tested thoroughly in the context of this thesis for the clang compiler version 4.0.0..

Address	Line	Column	File
0x400470	1	0	1
0x40047d	2	8	1
0x400481	2	10	1
0x400487	2	15	1
0x40048b	2	18	1
0x400491	2	22	1
0x400495	2	8	1
0x40049b	3	7	1
0x4004a7	5	7	1
0x4004ae	7	1	1
0x4004c0	9	0	1
0x4004c6	9	12	1
0x4004c8	9	12	1

Figure 4.3: Debugging information showing the mapping lines and columns of the source code from figure 4.1a and program addresses from object code from figure 4.4 (`llvm-dwarfdump -debug-dump=line a.out`).

```
1   0x400470 <+0>:   push   %rbp
2   0x400471 <+1>:   mov    %rsp,%rbp
3   0x400474 <+4>:   mov    %edi,-0x8(%rbp)
4   0x400477 <+7>:   mov    %esi,-0xc(%rbp)
5   0x40047a <+10>:  mov    %edx,-0x10(%rbp)
6   0x40047d <+13>:  cmpl   $0x0,-0x8(%rbp)
7   0x400481 <+17>:  jne    0x40049b <testFunction+43>
8   0x400487 <+23>:  cmpl   $0x3,-0xc(%rbp)
9   0x40048b <+27>:  jge    0x4004a7 <testFunction+55>
10  0x400491 <+33>:  cmpl   $0x5,-0x10(%rbp)
11  0x400495 <+37>:  jne    0x4004a7 <testFunction+55>
12  0x40049b <+43>:  movl   $0x1,-0x4(%rbp)
13  0x4004a2 <+50>:  jmpq   0x4004ae <testFunction+62>
14  0x4004a7 <+55>:  movl   $0x0,-0x4(%rbp)
15  0x4004ae <+62>:  mov    -0x4(%rbp),%eax
16  0x4004b1 <+65>:  pop    %rbp
17  0x4004b2 <+66>:  retq
```

Figure 4.4: Object code, compiled from the code of figure 4.1a for x86-64 architecture.

For example, in figure 4.4 the conditional jump `jne` in line seven corresponds to the condition a and therefore has no relational operator. Table 4.1 tells that in this case a detected jump means that the condition has been evaluated as **True**. However the same kind of conditional jump in line eleven corresponds to the condition $c == 5$. Because of that detecting the execution of this jump implies that the condition has been evaluated as **False**.

relational operator:	architecture:		interpretation of jump:
	x86-64	ARM	
no operator	`jne`	`bne`	**True**
$(! = 0)$	`je`	`beq`	**False**
$==$	`je`	`beq`	**True**
	`jne`	`bne`	**False**
$<$	`jl`	`blt`	**True**
	`jge`	`bge`	**False**
$<=$	`jle`	`ble`	**True**
	`jg`	`bgt`	**False**
$>$	`jg`	`bgt`	**True**
	`jle`	`ble`	**False**
$>=$	`jge`	`bge`	**True**
	`jl`	`blt`	**False**

Table 4.1: Multiple interpretations of jumps in the x86-64 and ARM instructions sets compiled with clang version 4.0.0.

4.3 Assignment Reconstruction

Knowing the conditional jumps that correspond to conditions in the source code, makes it possible to reconstruct the assignment of the conditions during an execution by analyzing the performed jumps. For this analysis watchpoints can be declared. These watchpoints are the program address of each conditional jump and the target address of the jump. If that following program address equals the jump that is described in the conditional jump instruction, a jump has been performed otherwise it has not. Table 4.1 shows how the information, whether a jump has been performed can be used to reconstruct the assignment of the condition. In chapter 6 is explained how the watchpoints and other information about the conditions can be used to generate online monitors.

condition:	relational operator:	instruction:	interpretation of jump:
a	none	`jne`	**True**
$b < 3$	$<$	`jge`	**False**
$c == 5$	$==$	`jne`	**False**

Table 4.2: Example interpretation of detected jumps.

5 Offline MC/DC Measurement

After the interesting program addresses (watchpoints) are declared by static analysis they can be used to analyze program traces that contain the sequence of all executed instruction during the execution of a program. In order to measure MC/DC the assignments of the conditions can be reconstructed and fill the MC/DC-table. MC/DC-table in this context means all cases based on the reconstructed assignments and the outcome of the decision (see figure 5.1). Because the outcome of the decision is needed to evaluate the independent effect of a condition, it has to be be determined by evaluating the decision with the reconstructed assignments. The coverage can then be measured by iterating through the cases and showing the independence of each condition by finding independent pairs. A general overview of this approach is provided in figure 5.1.

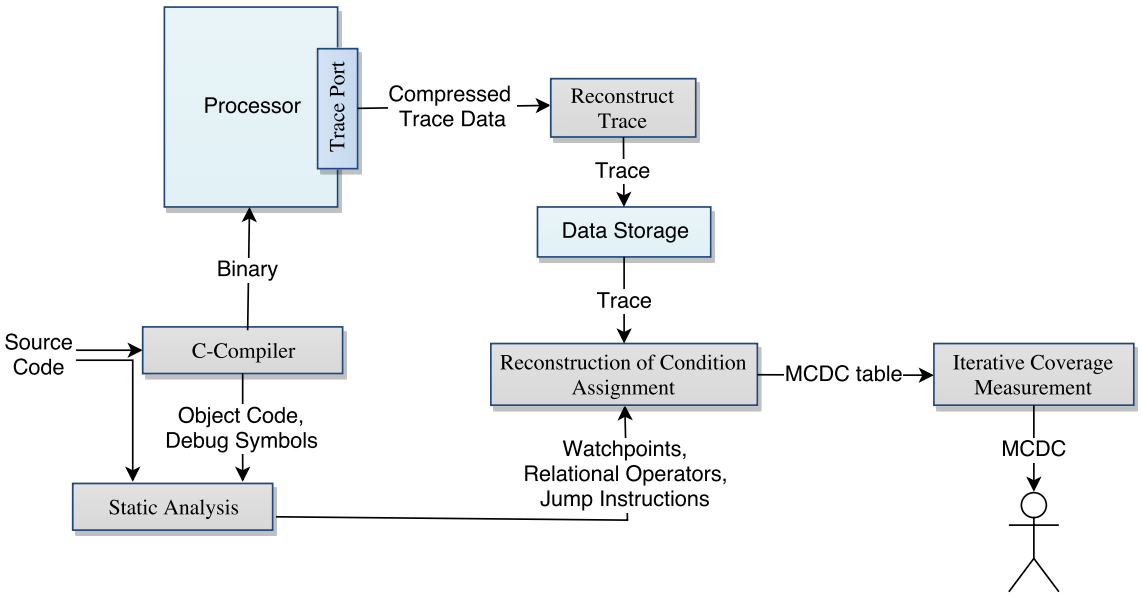


Figure 5.1: General overview of offline MC/DC measurement.

The static analysis is described in chapter 4 and needs access to the source code and the object code including debug symbols. The trace can be gathered through the trace port of the processor that is executing the program and can be saved in a data storage after the reconstruction process. IntelPT[Rei13] and ARM DSTREAM

[ARM17] are technologies that are capable of doing this. The assignment reconstruction needs the program addresses (watchpoints) defined by the static analysis as well as the information about the relational operators and jump instructions to analyze the recorded trace. After the assignments of the conditions are reconstructed, the MC/DC table can be filled and MC/DC can be measured iteratively as is explained in the following.

5.1 Evaluation of Independence

	evaluated:			decision:
Nr.	A	B	C	A || (B && C):
1	F	F	?	F
2	F	T	F	F
3	F	T	T	T
4	T	?	?	T

Table 5.1: Example MC/DC table for the the decision A || (B && C).

The information needed for MC/DC evaluation are the different assignments of the conditions and the outcome of each decision in the analyzed program. In the case of short-circuit evaluation, which is assumed in the following, it is useful to utilize a three valued logic as described in section 3.2. To measure the coverage of a single decision each case is compared to all other cases. If there is only one condition (apart from not evaluated conditions) and the outcome of the decision differs, an independent pair for this condition has been found.

For example in table 5.1 the independent effect of condition C can be shown with test case nr. 2 and 3, because the conditions A (F) and B (T) are the same and only the condition C and the outcome of the decision are changing. In the case that some conditions are not evaluated, the test cases can still be used to show independence. For example test case nr. 1 and 4 can be used to show the independent effect of condition A, because A and the outcome are changing and the other conditions are either not evaluated at all or at least were not assigned with the same value.

After all entries have been compared with each other, the decision is considered MC/DC-covered, if for every condition at least one independent pair has been found. Because all test cases have to be compared with each other and every comparison consists of comparing each condition and the outcome, the runtime is quadratic with respect to the number of test cases. See section 7.3 for an implementation of this approach with Python.

5.2 Limits of the Offline Approach

The approach described in this chapter is relatively simple and can be implemented for various technologies like Intel® PT[Rei13] and ARM DSTREAM[ARM17]. Compared to the approach described in chapter 6 it has the advantage of working without knowledge about the structure of the decisions. However it contains a bottleneck, because the trace data that has to be stored in some form of storage space. The data rate of the trace information is so high that even very high speed storage can only record a few seconds of program execution.

ARM DSTREAM provides a trace buffer of 4 GB for a recording speed of 10 Gbit/s which leads to a possible trace recording of less than four seconds [DGH+17].

The traces generated by Intel® PT become big very quickly as well. For example a CPU benchmark program, that sums up all Integers from 0 to 100 million, takes less than 0.5 ms on an Intel i3-5005U CPU and generates a reconstructed Intel® PT trace that is 3.7 GB.

This limitation can be overcome by online reconstruction and monitoring as described in the following chapter.

6 Online MC/DC Measurement

To overcome the limitations of MC/DC measurement by analyzing recorded traces offline in the following an approach is proposed to measure MC/DC online by using FPGA hardware to reconstruct and filter the trace. This challenging task can be divided into smaller subproblems that are shown in figure 6.1. Besides the declaration of watchpoints by static analysis that has been explained in chapter 4 the structure of the decisions can be used to generate online monitors that reconstruct the assignments and can evaluate MC/DC by reconstructing the test cases.

The approach, that is described in the following, has not been realized, because the technology is currently under development and has not been finished during the work on this thesis. In order to show the the general feasibility the traces are generated with Intel® PT and the software back-end of TeSSLa [Leu18]. These parts are replacements for the trace generation with online trace reconstruction and the monitoring with FPGA hardware.

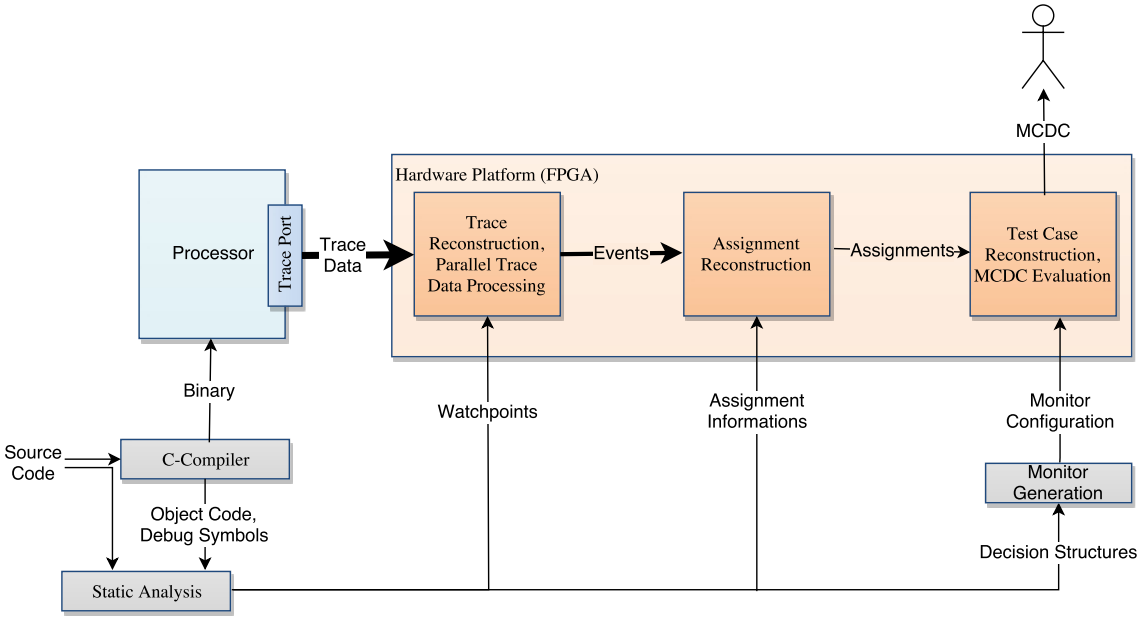


Figure 6.1: General overview of the proposed method of measuring MC/DC online using trace data reconstruction and event monitoring.

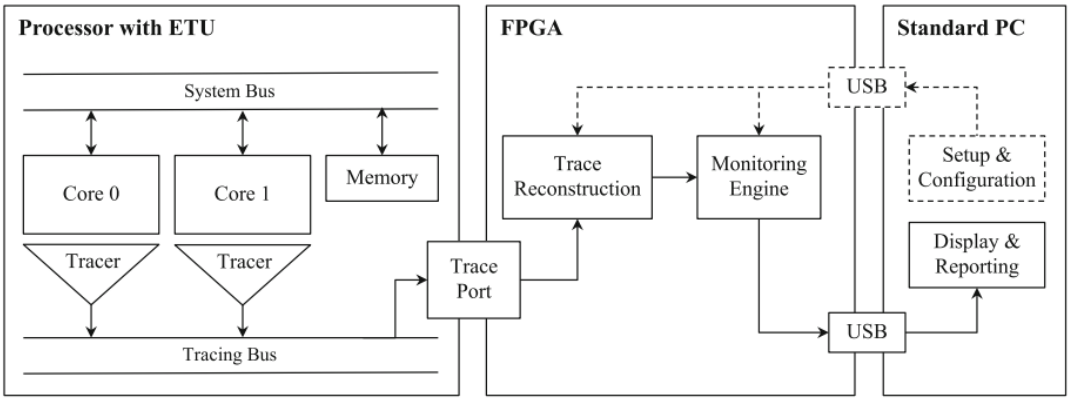


Figure 6.2: Overview of the trace data reconstruction setup[DGH+17].

6.1 Online Trace Monitoring

The approach to implement both trace reconstruction and monitoring using FPGA hardware is currently under research and described in [DGH+17]. In advantage of prior hardware-supported monitoring frameworks it is possible to use a standard product, i.e. ARM CoreSight, and to rapidly adjust the monitoring system by using a flexible configurable FPGA-based event processing platform.

Figure 6.2 shows how a multi core processor provides trace data that can be reconstructed using FPGA hardware. The cores are communicating through the system bus and simultaneously sending trace data through the trace bus to the trace port. This setup with two separated buses allows capturing trace data without affecting the behavior of the cores or other system components. The trace data, which comes in a highly compressed format, is then reconstructed using a *Program Flow Trace*-based method implemented in FPGA hardware. Prior to the reconstruction process the binary is statically analyzed and a lookup table for all targets of the conditional jumps is stored on the FPGA. Because the data rate of the reconstructed trace would be extremely high, watchpoints (or *tracepoints*) can be set in order to filter events with a certain reconstructed address. This filtering process can be part of the reconstruction process by immediately comparing the reconstructed address to the address of the predefined watchpoints. This can be achieved by setting an additional tracepoint flag to the lookup table[DGH+17].

For the evaluation of MC/DC these tracepoints equal the watchpoints that were gathered during the static analysis (see chapter 4).

6.2 Assignment Reconstruction

The static analysis described in chapter 4 provides program addresses of the conditional jumps and their target jump address. When an event with the program address of a conditional jump arrives, the following executed program address has to be checked in order to find out whether the jump has been taken. The assignment then can be reconstructed by interpreting what the meaning of that jump is (see section 4.3 for details).

6.3 Generate Online Monitors for MC/DC

The reconstructed assignments can be recorded and evaluated in the same way as described in chapter 5. However it is now possible to build a monitor to evaluate MC/DC online. The advantage is that no matter how much time the execution takes, no storage space is needed and the only output is the coverage.

6.3.1 Determining all possible Cases

In order to efficiently determine MC/DC by analyzing traces online it is necessary to detect the structure of the decisions before running the program and to determine all possible cases of assignments these decisions can have. As stated in section 3.2 there are less distinguishable cases when short-circuit evaluation is assumed. With the before reconstructed assignments it is possible to detect which test case was executed.

The distinguishable cases can be determined by analyzing the structure of the decision and testing all possible input combinations while tracking which of those were evaluated at all. For the generation of MC/DC monitors the cases must be enumerated.

6.3.2 MC/DC Monitors

After determining all distinguishable cases it is then possible to declare which of these are needed to show the independence of certain conditions. Deterministic finite automatons (DFA) can be used to describe which cases are needed to fulfill MC/DC. The input symbols are the possible assignments a single test case can produce. MC/DC is fulfilled as soon as the accepting state is reached. Figure 6.3 shows an example DFA that accepts, if the adequate assignments for MC/DC of the

decision $A\&\&B$ is a subset of the input. Every possible input sequence has to be considered during the creation of the DFA. That means that for a decision with n distinguishable test cases $n!$ sequences have to be analyzed for MC/DC. Because of that immense growth of paths in the DFA they are not feasible for decisions with more than three conditions.

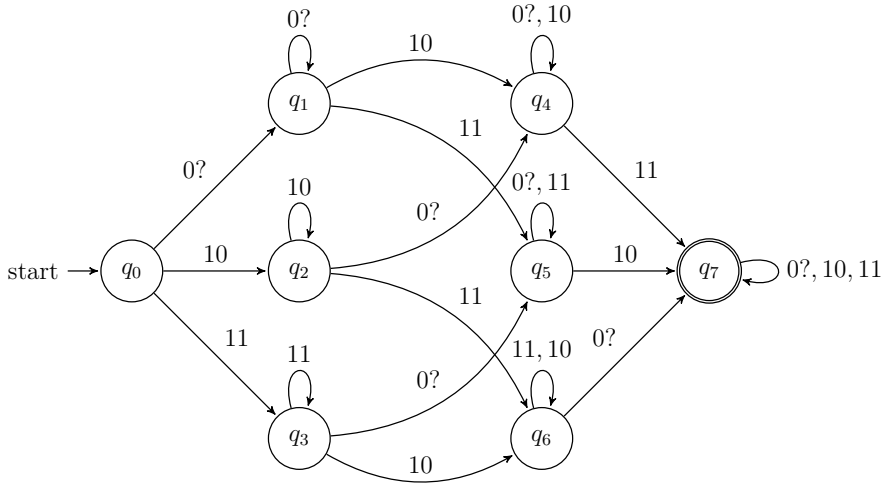


Figure 6.3: DFA for MC/DC evaluation of the decision $A\&\&B$.

To overcome this problem a more efficient approach has been developed utilizing the fact that for the evaluation of MC/DC it is necessary to record the incoming cases, but not their order. The information about, what cases have already been seen, can be stored efficiently as a set. If a case is detected, it will be added to the set.

In order to evaluate MC/DC this set can be checked against a logical formula that can be calculated by analyzing the decision. Chapter 3 explains how short-circuit evaluation reduces the number of possible cases and how the independent pairs for each condition in a decision can be found. These independent pairs are now used to calculate the logical formula. For every condition in a decision there is at least one independent pair needed to show this conditions independence. That idea leads to a formula that is a conjunction of all possible ways to show independence of each condition.

Because in the case of the in chapter 3 discussed decision A || (B && C) all cases are needed to show MC/DC a more complex decision is introduced in table 6.1 that helps to understand the benefit of using logical formula to evaluate MC/DC. To show for example the independence of the condition A in (A&&B) || (C && D) one of the two independent pairs $1, 7$ and $2, 7$ have to be shown, which can be expressed as a logical formula: $(case1 \wedge case7) \vee (case2 \wedge case7)$. The complete logical formula for MC/DC can be constructed with a conjunction over all conditions, because the

	evaluated:				decision:	pairs:			
Nr.	*A*	*B*	*C*	*D*	(A&&B) || (C && D):	*A*	*B*	*C*	*D*
1	F	?	F	?	F	7		3	
2	F	?	T	F	F	7			3
3	F	?	T	T	T			1	2
4	T	F	F	?	F		7	6	
5	T	F	T	F	F		7		6
6	T	F	T	T	T			4	5
7	T	T	?	?	T	1,2	4,5		

Table 6.1: All distinguishable cases and independent pairs for the decision (A&&B) || (C && D).

independence has to be shown for all conditions. For the example decision shown in table 6.1 the complete formula can be expressed as follows:

$$((case1 \wedge case7) \vee (case2 \wedge case7)) \wedge$$
$$((case4 \wedge case7) \vee (case5 \wedge case7)) \wedge$$
$$((case1 \wedge case3) \vee (case4 \wedge case6)) \wedge$$
$$((case2 \wedge case3) \vee (case5 \wedge case6))$$

See section 7.4 for details how this can be implemented as a monitor with a stream-based language.

7 Implementations

In the following chapter is described how the ideas, that have been previously introduced, can be implemented in order to show the feasibility of MC/DC measurement by analyzing program traces.

7.1 Trace Generation

There are multiple ways to obtain program traces from the CPU. Because some of them need dedicated hardware or were not accessible during the work on this thesis, a GDB script has been used to simulate a trace for training purposes. GDB is the *GNU Project Debugger* and is usually used to find bugs by setting breakpoints and analyzing the execution of program. The GDB script executes a program and prints each executed steps to a log file. This has shown to work on Intel and ARM processors.

Intel Processor Trace (Intel PT) is a feature of modern Intel CPUs and allows low-overhead execution tracing. It works by capturing information about program execution on each hardware thread using dedicated hardware facilities so that after execution completes software can do processing of the captured trace data and reconstruct the exact program flow[Rei13]. Traces generated with IntelPT have shown to work with the offline approach and as input for the simulation of the online approach.

7.2 Static Analysis

The static analysis aims as described in chapter 4 to find decision and conditions in the source code and to map each condition to a conditional jump in the object code. The analysis of the source code can be done by accessing the Abstract Syntax Tree (AST) provided by Clang[Tea17a].

7.2.1 LibClang

There are multiple possibilities how to access the AST. The most comfortable way is using the Python-bindings of *LibClang*, a high level C interface to Clang.

With LibClang it is possible to describe cursors that equal nodes in AST. These cursors have attributes like their *CursorKind*, that can be used to identify i.e. if statements (see figure 7.1 line 3).

In line 14 can be seen how new cursors, that are children of other cursors can be created recursively in order to find nested statements.

```
1  def find_if_stmts(cursor, if_stmts):
2    stmt = IfStmt()
3    if cursor.kind == IF_STMT:
4      stmt.add_if_location(parse_extent_info(str(cursor.extent)))
5      for i in cursor.get_children():
6        if i.kind == COMPOUND_STMT:
7          stmt.add_then_location(parse_extent_info(str(i.extent)))
8
9        if i.kind == BINARY_OPERATOR or i.kind == UNEXPOSED_EXPR:
10         stmt.add_decision_location(parse_extent_info(str(i.extent)))
11
12       if_stmts.append(stmt)
13
14   for c in cursor.get_children():
15     find_if_stmts(c, if_stmts)
16
17   return if_stmts
```

Figure 7.1: LibClang implementation for finding if statements. Parts of this code are simplified in order to be more precise.

LibClang works fine for finding simple if statements, but unfortunately LibClang does not provide full control of the AST, which is necessary to find more complicated decision that are for example hidden in assignment statements (see section 3.3).

7.2.2 LibTooling/LibASTMatchers

LibTooling is a C++ interface that makes it possible to write tools that have full access over the AST. The recently introduced *LibASTMatcher*[Tea17b] can be used alongside with LibTooling and simplifies the specification of patterns that the AST is supposed to match. In a domain specific language can be expressed what kind of nodes are searched and which information should be extracted. A simple example is provided in figure 7.2.

```
1 StatementMatcher IfMatcher =
2   anyOf(
3     ifStmt().bind("if_stmt"),
4     whileStmt().bind("while_stmt")
5     );
```

Figure 7.2: LibASTMatcher for finding if and while statements in C code.

7.2.3 Reading Debugging Symbols

After finding the decisions and conditions the DWARF[E+07] debugging information has to be analyzed in order to map each condition to a conditional jump in the object code. To enable the debugging information including a mapping between lines/columns in the source code and addresses in the object when using the Clang compiler the following flags can be set: `-g -Xclang -dwarf-column-info`

pyelftools is a Python library that provides parsing and analyzing DWARF debugging information. Figure 7.3 shows how it can be implemented. *get_dwarf_info()* returns a DWARFInfo context object, which is the starting point for processing in pyelftools. It can be iterated to gather all the location information in the *.debug_info* section.

```
1  def map_line_column(filename):
2    line_column_address = []
3    with open(filename, 'rb') as f:
4      elffile = ELFFile(f)
5      dwarfinfo = elffile.get_dwarf_info()
6
7      for CU in dwarfinfo.iter_CUs():
8        lineprog = dwarfinfo.line_program_for_CU(CU)
9        for entry in lineprog.get_entries():
10         line_column_address.append((entry.state.line,
11                                     entry.state.column, hex(entry.state.address)))
12
13   lines_set = set(line_column_address)
14   sorted_mapping = sorted(lines_set, key=lambda tup: (tup[0], tup[1]))
15   return sorted_mapping
```

Figure 7.3: Find and sort the mapping between locations in the source code and addresses in the object code.

7.3 Iterative MC/DC Evaluation

Measuring MC/DC with the offline approach described in chapter 6 is done by finding all independence pairs in a set of test cases. This can be done by iteratively comparing all test cases with each other and find out if only one condition and the outcome of the decision is changing.

Figure 7.5 shows an implementation with Python. *measure_mcdc()* gets the reconstructed test cases as vectors with possible entry values of 0, 1, 2 (encoded for **False**, **?**, **True**). The last entry of each vector has the value of the outcome of the decision.

By iterating over all test cases they are compared to each other by calling the function *eval_independence* with the pair that is supposed to be evaluated. This function can be seen in figure 7.4. First is checked, if the outcome of the two test cases are the same, because if they are, they cannot show the independent effect of any condition (line 2).

After that all condition that change are counted. Note that the conditions, that were not evaluated in one of the test cases are not counted (line 7). If the amount of changing condition is exactly 1, this pair of test cases are independence pairs for this condition and the condition is returned. Otherwise -1 is return to indicate that no independence pair was found.

With this implementation it is not only possible to evaluate, if a set of test cases fulfill MC/DC, but also which conditions are missing in case that MC/DC is not fulfilled.

```
1  def eval_independence(vector_double):
2    if vector_double[0][-1] == vector_double[1][-1]:
3      return -1
4
5    counter = 0
6    for i in range(0,len(vector_double[0])-1):
7      if vector_double[0][i] == 1 or vector_double[1][i] == 1:
8        continue
9      if vector_double[0][i] != vector_double[1][i]:
10       counter += 1
11       ind_con = i
12   if counter == 1:
13     return ind_con
14   else:
15     return -1
```

Figure 7.4: Evaluating if a pair of test cases shows the independent effect for a condition.

```
1  def measure_mcdc(test_cases):
2    ind_cons = []
3    for i in range(0,len(test_cases)):
4      for j in range(i+1,len(test_cases)):
5        if eval_independence([test_cases[i],test_cases[j]]) != -1:
6          ind_cons.append(eval_independence([test_cases[i],test_cases[j]]))
7
8    missing_set = set(range(0,len(test_cases[0])-1)) - set(ind_cons)
9    return missing_set
```

Figure 7.5: Iterating through the test cases and call *eval_independence* with every test case pair.

```
1 #include "CuTest.h"
2 #include <stdio.h>
3
4
5 int exampleFunction(int a, int b, int c) {
6
7     if (a || (b && c)){
8         return 1;
9     }
10    if (a<2 && b){
11        return 0;
12    }
13    return 0;
14 }
15
16 void TestExampleFunction(CuTest *tc) {
17   int actual = exampleFunction(0, 0, 0);
18   CuAssertIntEquals(tc, 0, actual);
19
20   actual = exampleFunction(0, 1, 0);
21   CuAssertIntEquals(tc, 0, actual);
22
23   actual = exampleFunction(0, 1, 1);
24   CuAssertIntEquals(tc, 1, actual);
25
26   actual = exampleFunction(1, 1, 1);
27   CuAssertIntEquals(tc, 1, actual);
28 }
29
30 CuSuite* mathUtilGetSuite() {
31    CuSuite* suite = CuSuiteNew();
32    SUITE_ADD_TEST(suite, TestExampleFunction);
33    return suite;
34 }
```

Figure 7.6: Example C code containing simple unit testing of the function *exampleFunction*.

```
1 Parse Source Code...
2 Process Debug Informations...
3
4 Found Decision:    a || (b && c)
5 Found Condition:   ('a', 7, 9)
6 Found Condition:   ('b', 7, 15)
7 Found Condition:   ('c', 7, 20)
8
9 Reconstruct Assignment with trace gdb.log :
10 [['a', 0], ['b', 0], ['c', 1]]   Outcome: 0
11 [['a', 0], ['b', 2], ['c', 0]]   Outcome: 0
12 [['a', 0], ['b', 2], ['c', 2]]   Outcome: 2
13 [['a', 2], ['b', 1], ['c', 1]]   Outcome: 2
14 This Decision is MC/DC-covered.
15
16 ############################
17
18 Found Decision:    a<2 && b
19 Found Condition:   ('a<2', 10, 10)
20 Found Condition:   ('b', 10, 16)
21
22 Reconstruct Assignment with trace gdb.log :
23 [['a<2', 2], ['b', 0]]   Outcome: 0
24 [['a<2', 2], ['b', 2]]   Outcome: 2
25 The independence of the condition ('a<2', 10, 10) cannot be shown based on the test cases.
26
27 ############################
28
29 MC/DC Coverage: 50.0%
```

Figure 7.7: Output of the implementation of the offline approach using a gdb trace and the C code in figure 7.6.

Figure 7.7 shows the output of the implementation of the offline approach. The coverage of a simple C program, that is shown in figure 7.6, is measured. The C code contains a function, that is called multiple time with a unit test library called *CuTest*.

It can be seen that two decisions are detected and the assignments of the conditions are reconstructed by analyzing a gdb trace. While the first decision has sufficient test cases for MC/DC coverage, the second decision is missing a test case to show the independent effect of the condition "a<2", which is indicated in line 25. The MC/DC coverage of 50.0% is calculated by dividing all covered decisions by all detected decision inside the C code.

7.4 Online Monitors

The online approach described in chapter 6 was implemented as part of this thesis by using the software interpreter of TeSSLa and trace generated with gdb and Intel PT, because the trace reconstruction hardware and the TeSSLa hardware interpreter was not available yet. Figure 7.8 shows, which part of the online approach have been implemented.

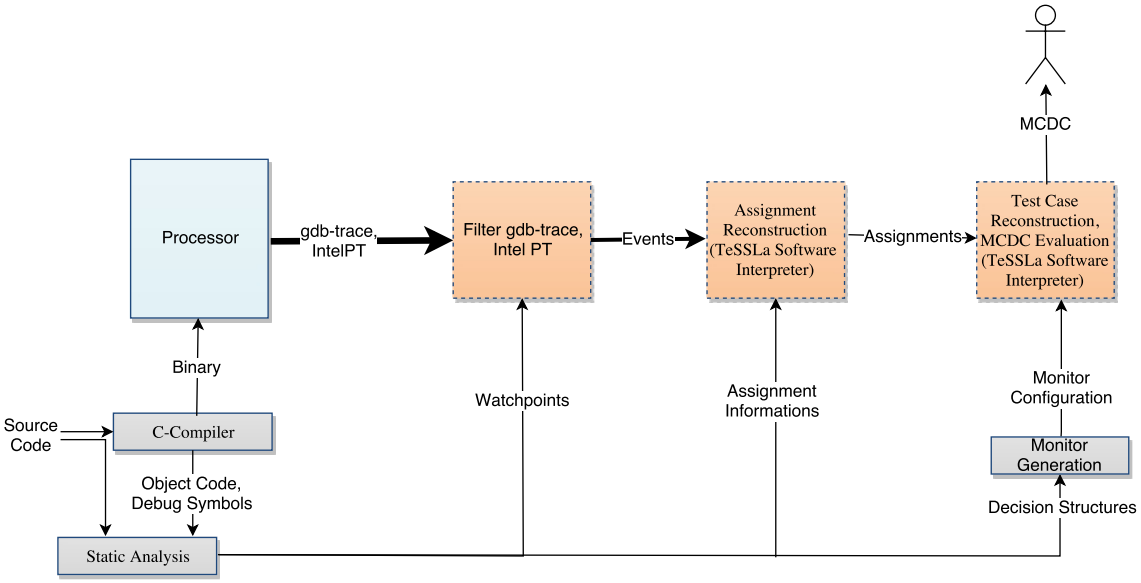


Figure 7.8: General overview of which part of the online approach (see figure 6.1) were simulated as part of this thesis.

7.4.1 TeSSLa

The Temporal Stream-Based Specification Language (short TeSSLa) is designed for handling large amounts of trace data. TeSSLa reasons over asynchronous input streams and new streams can be derived by applying functions to the input streams or other already derived internal streams[DGH+17].

TeSSLa has primitives that are necessary to express typical challenges with stream-based specifications[Leu18] and are used in the implementations:

- "The *last* operation takes two steams and returns the previous value of the first stream at the time stamps of the second."
- "The *default* operation adds a value at time zero to a stream if no value is present."

- "The *lift* operation lifts a n-ary function f from values to streams." Lifted functions that are available are i.e. Boolean functions (and, or) and Integer functions (+,-, bit shift, comparison).

7.4.2 Assignment Reconstruction

TeSSLa can be used to describe how the assignment reconstruction described in section 6.2 can be performed. In figure 7.9 an example is shown where the conditional jump has the address 4200801 and the corresponding jump target address is 4200827 and a performed jump is interpreted as an evaluation as **True**.

Because of the three valued logic **True** is encoded as 2 and **False** is encoded as 0. If the the condition has not been evaluated, the default value of 1 is given. If a detected jump is interpreted as an assignment of **True** or **False** has to figured out during the static analysis (see table 4.1).

The function *mergeValues()*, because the stream $c0$ would otherwise be set to the reconstructed Value only for one step. *mergeValues()* makes it possible to set a stream to value depending on a condition. This value can only be changed again, if another condition is true.

```
1 def c0:= default(
2   mergeValues(
3     if last(pc, pc) == 4200801 && pc == 4200827 then 2,
4     if last(pc, pc) == 4200801 && pc != 4200827 then 0
5   ),-1)
6
7 def mergeValues(a, b):=
8   if default(time(a), 0) >= default(time(b), 0) then default(a, b)
9   else default(b, a)
```

Figure 7.9: Condition reconstruction example described with TeSSLa.

7.4.3 Test Case Reconstruction

After figuring out how the conditions where assigned by analyzing the jumps, they can be used to reconstruct which test case was executed. The example in figure 7.10 shows the test case reconstruction for the simple decision $A\&\&B$, which possible test cases can be seen in table 7.1. Note that the condition B is not relevant, if the condition A is evaluated as **False** and therefore B is not evaluated at all due to short-circuit logic.

Nr.	*A*	*B*	(A && b):
0	F	?	F
1	T	F	T
2	T	T	T

Table 7.1: All possible test cases for decision (A && B).

```
1 def caseNumber :=
2   if c0 == 0 then 0
3   else if c0 == 2 && c1 == 0 then 1
4   else if c0 == 2 && c1 == 2 then 2
5   else -1
```

Figure 7.10: Test case reconstruction example described with TeSSLa.

7.4.4 Monitoring

As described in section 6.3 online MC/DC monitoring can be implemented using sets, which can be checked with a logical formula in order to evaluate MC/DC. With TeSSLa the sets can be implemented as bit sets, where every bit represents one predefined case as shown in figure 7.11. Because TeSSLa is stream-based, a set has to be specified recursively with the *last()* operator, because a stream-based set is always depending on the state of the set in the last step. The start value of the recursion is defined with the *default()* operator. To add a new entry the function *set_add()* is called, that uses a bit shift operator to set the position in the Integer, where the entry should be stored.

```
1 def set := default(set_add(last(set, caseNumber), caseNumber), 0)
2
3 def set_add(set, x) :=  if in_set(set, x) then set
4   else set + (1 << x)
```

Figure 7.11: Defining a set and adding every case that is detected.

In order to check if MC/DC is fulfilled for a decision with the detected cases, the logical formula can be described with TeSSLa as shown in figure 7.12 for the example decision (A&&B) || (C && D). The function *in_set()* checks if an entry is part of a bit set by using a shifted Integer containing a 1 at the place of the entry and comparing it to the set with a bitwise AND operator. The generated stream *mcdc* is true as soon as there are a set of test cases discovered that satisfy MC/DC.

The logical formula from section 6.3.2:

$$((case1 \wedge case7) \vee (case2 \wedge case7)) \wedge$$
$$((case4 \wedge case7) \vee (case5 \wedge case7)) \wedge$$
$$((case1 \wedge case3) \vee (case4 \wedge case6)) \wedge$$
$$((case2 \wedge case3) \vee (case5 \wedge case6))$$

is implemented in figure 7.12

```
1 def mcdc := ((in_set(set,1) && in_set(set,7))||(in_set(set,2) && in_set(set,7))) &&
2             ((in_set(set,4) && in_set(set,7))||(in_set(set,5) && in_set(set,7))) &&
3             ((in_set(set,1) && in_set(set,3))||(in_set(set,4) && in_set(set,6))) &&
4             ((in_set(set,2) && in_set(set,3))||(in_set(set,5) && in_set(set,6)))
5
6 def in_set(set, x) :=   set & (1 << x) > 0
```

Figure 7.12: Checking MC/DC for the decision (A&&B) || (C && D) by evaluation a logical formula over cases in a set.

Implementing the online approach with TeSSLa and simulating it with the TeSSLa software interpreter has shown the feasibility of the concept. It has to be further investigated how this can work with the TeSSLa hardware interpreter and with FPGA hardware.

8 Conclusion

This work presents two approaches how MC/DC can be measured by using trace information as an alternative to state-of-the-art solutions like software instrumentation. Showing that MC/DC is generally not implied by object branch coverage made clear that the only way to measure MC/DC based on traces is to figure out how the condition were assigned during the execution. This can be accomplished by finding out which conditions in the source code correspond to which conditional jumps in the object code. With the addresses of the conditional jumps and their jump addresses it is possible to reconstruct how conditions were assigned during execution. Because most modern programming languages use short-circuit evaluation, it had to be shown how MC/DC can still be measured, when conditions were not assigned at all. The results are that MC/DC with short-circuit evaluation is equivalent to Masking MC/DC, which is accepted in avionic industry by the DO-178C.

The main contribution of this work is the concept of what information is needed and how source and object code can be analyzed to gather the needed information. Furthermore it describes how this information can be used to reconstruct assignments of conditions by evaluating the trace. The reconstructed assignments can be used to fill a MC/DC-table and to measure MC/DC offline. Alternatively the structure of the decisions can be used to build monitors that analyze traces online and therefore have no time limitations.

Overall measuring MC/DC based on jumps is a promising concept that has been shown to work on simple examples with existing trace-technologies. However there are some limitations that lie within the concept and further effort has to be done before the idea can possibly used in practice.

8.1 Limits

Measuring MC/DC based on jumps has some general problems. If the compiler uses any optimization level, it is more than likely that conditions are not directly translated to conditional jumps. It is possible that conditional moves, jump tables or indirect branches are used. Program traces deliver no information how these instructions are evaluated and so non of these structure can be used to reconstruct condition by analyzing the trace.

8.2 Outlook

The offline approach has been shown to work with IntelPT and small example programs. It was not possible to try the online approach, because the technology was not available during the work of this thesis and was only simulated as part of this work. Online approach should be implemented using the FPGA hardware as soon as the development of FPGA-based trace reconstruction is finished in order to make MC/DC measurement possible for long term program executions.

As stated in section 4.1.2 it has to be further investigated how statement assignment that contain Boolean expressions and later are used in branch points can be detected with static analysis.

As stated in section 3.5 coverage analysis on object code level complies with the DO-178C only if the same level of assurance is guaranteed as coverage analysis on the source code level. This requirement has to be further investigated.

The Python implementations are a good start point to develop are tool that can be used more easily and has some form visualization in order to show the detected and covered decisions directly in the source code. Software, that is actually used in the avionic industry, should be used to evaluate this approach from a practical perspective.

List of Figures

List of Tables

Abbreviations

MC/DC	Modified condition/decision coverage
ETU	Embedded trace units
FPGA	Field programmable gate array
DO-178C	Software Considerations in Airborne Systems and Equipment Certification
CAST	Certification authorities software team
OBC	Object branch coverage
BDD	Binary decision diagram
AST	Abstract syntax tree
DFA	Deterministic finite automaton
TeSSLa	Temporal Stream-Based Specification Language

Bibliography

[ARM13] ARM Limited (Hrsg.): *ARM IHI 0029B: CoreSightTM Architecture Specification v2.0, issue D.* ARM Limited, 2013

[ARM17] ARM Limited (Hrsg.): *DS-5 ARM DSTREAM User Guide Version 5.27.* ARM Limited, 2017

[BCG+10] Bordin, Matteo ; Comar, Cyrille ; Gingold, Tristan ; Guitton, Jérôme ; Hainque, Olivier ; Quinot, Thomas: Object and source coverage for critical applications with the couverture open analysis framework. In: *ERTS (Embedded Real Time Sofware and Systems Conference)*, 2010

[Bha07] Bhansali, Praful V.: The MCDC paradox. In: *ACM SIGSOFT Software Engineering Notes* 32 (2007), Nr. 3, S. 1–4

[CAS03] CAST: Structural Coverage of Object Code, Position Paper 17 / Certification Authorities Software Team. 2003. – Forschungsbericht

[CGHQ12] Comar, Cyrille ; Guitton, Jerome ; Hainque, Olivier ; Quinot, Thomas: Formalization and comparison of MCDC and object branch coverage criteria. In: *ERTS (Embedded Real Time Software and Systems Conference)*, 2012

[Chi01] Chilenski, John J.: An investigation of three forms of the modified condition decision coverage (MCDC) criterion / BOEING COMMERCIAL AIRPLANE CO SEATTLE WA. 2001. – Forschungsbericht

[CK07] Chilenski, J.J. ; Kurtz, J.L.: Object-Oriented Technology Verification Phase 3 Report—Structural Coverage at the Source and Object Code Levels / FAA report DOT/FAA/AR-07/20. 2007. – Forschungsbericht

[CM94] Chilenski, John J. ; Miller, Steven P.: Applicability of modified condition/decision coverage to software testing. In: *Software Engineering Journal* 9 (1994), Nr. 5, S. 193–200

[DGH+17] Decker, Normann ; Gottschling, Philip ; Hochberger, Christian ; Leucker, Martin ; Scheffel, Torben ; Schmitz, Malte ; Weiss, Alexander: Rapidly Adjustable Non-intrusive Online Monitoring for Multi-core Systems. In: *Brazilian Symposium on Formal Methods* Springer, 2017, S. 179–196

[E+07] Eager, Michael J. u. a.: Introduction to the dwarf debugging format. In: *Group* (2007)

[HVCR01] Hayhurst, Kelly J. ; Veerhusen, Dan S. ; Chilenski, John J. ; Rierson, Leanna K.: A practical tutorial on modified condition/decision coverage. (2001)

[KC15] Kandl, Susanne ; Chandrashekar, Sandeep: Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation. In: *Computing* 97 (2015), Nr. 3, S. 261–279

[KF99] Kaner, Cem ; Falk, Jack: *Testing computer software*. Wiley, 1999

[LA04] Lattner, Chris ; Adve, Vikram: LLVM: A compilation framework for lifelong program analysis & transformation. In: *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization* IEEE Computer Society, 2004, S. 75

[Leu18] Leucker, Sanchez C. Scheffel T. Schmitz M. Schramm A. M.: TeSSLa: Runtime Verification of Non-synchronized Real-Time Streams. In: *ACM Symposium on Applied Computing (SAC)*, 2018. – to appear

[Mye06] Myers, Glenford J.: *The art of software testing*. John Wiley & Sons, 2006

[Pot12] Pothon, Frederic: DO-178C/ED-12C versus DO-178B/ED-12B: Changes and Improvements. In: *Frederic Pothon* (2012)

[Rei13] Reinders, James: *Processor Tracing*. Website, 2013. – Online available `https://software.intel.com/en-us/blogs/2013/09/18/processor-tracing`, accessed: 29.12.2017.

[Rie13] Rierson, Leanna: *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. CRC Press, 2013

[T+01] Team, CAST u. a.: Rationale for accepting masking mcdc in certification projects. In: *Position Paper 6, Tech. Rep.* (2001)

[T+02] Team, FCAS u. a.: What is a" decision" in application of modified condition/decision coverage and decision coverage (dc). In: *Technical Report position paper* (2002)

[Tea] Team, The G.: *Options That Control Optimization*. Website, . – Online available `https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html`

[Tea17a] Team, The C.: *Clang 7 documentation*. Website, 2017. – https://clang.llvm.org/docs/Tooling.html

[Tea17b] Team, The C.: *Matching the Clang AST*. Website, 2017. – https://clang.llvm.org/docs/LibASTMatchers.html